

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DISSERTATION

LAYERED SAFE MOTION PLANNING FOR AUTONOMOUS VEHICLES

by

Chien-Liang Chuang

September 1995

Dissertation Advisor:

Yutaka Kanayama

Approved for public release; distribution is unlimited.

19960208 124

DTIC QUALITY INSPECTED 11

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED Doctoral Dissertation
4. TITLE AND SUBTITLE LAYERED SAFE MOTION PLANNING FOR AUTONOMOUS VEHICLES			5. FUNDING NUMBERS
6. AUTHOR(S) Chuang, Chien-Liang			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words) The major problem addressed by this research is how to plan a safe motion for autonomous vehicles in a two dimensional, rectilinear world. With given start and goal configurations, the planner performs motion planning which will lead a vehicle to achieve its task safely. During the planning, in addition to the safety consideration, motion's smoothness is also taken into account. The approach taken was to divide whole motion planning task into two layers. The top layer finds a global path by decomposing the free space into convex regions, then searching for an optimal global path class. The bottom layer performs local motion planning which further subdivides the planning problem into mid-portion and end-portion motion planning. The local motion planning is carried out region by region along the global path class. As results, simple motion instructions are generated for each region. For execution of planned motion, a software system, Model-based Mobile robot Language (MML-11), was developed. This easy-to-use robot language provides users a convenient tool to program their applications through its function library. The results of the research were implemented in MML-11 and tested on an experimental robot Yamabico-11 successfully.			
14. SUBJECT TERMS Robotics, autonomous vehicles, global path planning, mid-portion motion planning, end-portion motion planning, steering function, forerunner simulation, line tracking.			15. NUMBER OF PAGES 237
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited

LAYERED SAFE MOTION PLANNING FOR AUTONOMOUS VEHICLES

Chien-Liang Chuang
Lieutenant Colonel, The Republic of China Army
B.S., Chinese Military Academy, R.O.C., 1975
M.S., National Defense Management College, R.O.C., 1989

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

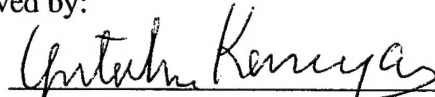
from the

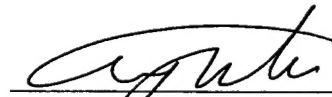
**NAVAL POSTGRADUATE SCHOOL
September 1995**

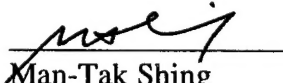
Author: _____

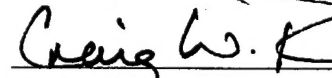

Chien-Liang Chuang

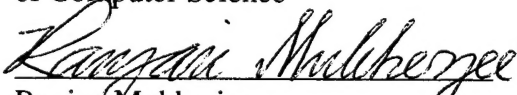
Approved by: _____


Yutaka Kanayama
Professor of Computer Science
Dissertation Supervisor


C. Thomas Wu
Associate Professor
of Computer Science


Man-Tak Shing
Associate Professor
of Computer Science

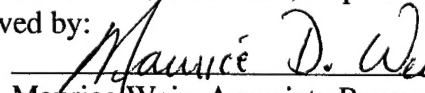

Craig W. Rasmussen
Assistant Professor
of Mathematics


Ranjan Mukherjee
Assistant Professor
of Mechanical Engineering

Approved by: _____


Ted Lewis, Chairman, Department of Computer Science

Approved by: _____


Maurice Weir, Associate Provost for Instruction

ABSTRACT

The major problem addressed by this research is how to plan a safe motion for autonomous vehicles in a two dimensional, rectilinear world. With given start and goal configurations, the planner performs motion planning which will lead a vehicle to achieve its task safely. During the planning, in addition to the safety consideration, motion's smoothness is also taken into account.

The approach taken was to divide whole motion planning task into two layers. The top layer finds a global path by decomposing the free space into convex regions, then searching for an optimal global path class. The bottom layer performs local motion planning which further subdivides the planning problem into mid-portion and end-portion motion planning. The local motion planning is carried out region by region along the global path class. As results, simple motion instructions are generated for each region.

For execution of planned motion, a software system, Model-based Mobile robot Language (MML-11), was developed. This easy-to-use robot language provides users a convenient tool to program their applications through its function library.

The results of the research were implemented in MML-11 and tested on an experimental robot Yamabico-11 successfully.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	REVIEW OF OTHER MOTION PLANNING APPROACHES	2
1.	Roadmap	2
2.	Cell Decomposition	3
3.	Potential Field	3
4.	Layered Approach	4
B.	SCOPE OF DISSERTATION	5
C.	MOTION PLANNING PROBLEM	5
1.	Problem Statement	5
2.	Layered Motion Planning	6
3.	Assumptions	8
D.	ORGANIZATION OF DISSERTATION	9
II.	GLOBAL PATH PLANNING	11
A.	HOMOTOPY	11
B.	OVERVIEW OF CONVEX POLYGONAL K-REGIONS THEORY ...	13
1.	K-decomposition	13
2.	Global Path Planning	15
III.	LOCAL MOTION PLANNING APPROACH	21
A.	PROBLEM STATEMENT	21
B.	PLANNING APPROACHES	22
1.	Break Down Planning	22
2.	Crossing Border with Orthogonal Orientation	24
C.	TERMINOLOGY	26
D.	MOTION PLANNING WITH STEERING FUNCTION	29

1.	Smooth Motion Planning with Curvature	29
2.	Steering Function	30
3.	Line Tracking with Steering Function	31
E.	FORERUNNER SIMULATION	33
1.	Overview	33
2.	Forerunner Simulation Structure	34
3.	Algorithms	36
F.	SYMMETRIC MOTION PLANNING AND REVERSE PATH	37
G.	REVERSE PATH TRACKING	40
H.	LOCAL MOTION PLANNING STEPS	44
1.	Determination of Crossing Position on the Border	45
2.	Orthogonalizing Border Orientations	47
3.	Algorithm	47
IV.	MID-PORION MOTION PLANNING	51
A.	INTRODUCTION	51
1.	Problem Statement	51
2.	Definitions	52
B.	STEERING FUNCTION CHARACTERISTICS	54
1.	Simulation Results Analysis	54
2.	Dynamic Smoothness	58
C.	PLANNING STRATEGIES	61
D.	TYPES OF COMBINED MOTIONS	62
E.	PLANNING INTRA-REGION MOTION OF TYPE I	64
1.	Region with Two Full-Borders	65
2.	Region with One Full-Border and One Partial-Border	65
3.	Region with Two Partial-borders	68

F.	PLANNING INTRA-REGION MOTION OF TYPE II	74
1.	Forward Length Greater than or Equal to Cross Length	74
2.	Forward Length Smaller than Cross Length	77
G.	PLANNING INTRA-REGION MOTION OF TYPE III	78
H.	MID-PORION MOTION PLANNING RULES	80
1.	Region with Two Parallel Borders	80
2.	Region with Two Perpendicular Borders	83
3.	Region with Borders on the Same Edge	84
I.	ALGORITHM	85
V.	END-PORION MOTION PLANNING	89
A.	PROBLEM STATEMENT	89
B.	PLANNING METHODS	89
1.	Forward Forerunner Simulation Application	90
2.	Collision Detection	93
3.	Backing-up Motion Simulation	95
C.	INITIAL MOTION PLANNING	98
1.	Determination of Reference Line	101
2.	Dynamic Smoothness in Initial Motion Planning	104
3.	Motion Planning Simulation	105
4.	Initial Motion Planning Algorithms	111
D.	FINAL MOTION PLANNING	115
VI.	YAMABICO HARDWARE ARCHITECTURE	119
A.	OVERVIEW	119
B.	IV-SPARC-33 CPU	119
1.	Address Map	121
2.	Registers	123

3.	Interrupt Map	123
4.	Internal Interrupts	123
5.	External Interrupts	124
C.	SONARS	124
1.	Sonar Control	125
2.	Sonar Grouping	126
VII.	MML-11 SOFTWARE ARCHITECTURE	127
A.	OVERVIEW	127
1.	System Architecture	127
2.	Interrupt-driven Subsystems	128
3.	Real Time Operating System	129
4.	User Program	129
B.	MOTION CONTROL ARCHITECTURE	129
C.	MOTION CONTROL SUBSYSTEM	131
1.	System Structure	131
2.	Transition from One Path to Another	133
D.	SUMMARY	139
VIII.	ROBOT REAL TIME OPERATING SYSTEM DESIGN	141
A.	INTRODUCTION	141
B.	MULTITASKING OPERATING SYSTEM DESIGN	141
1.	System Design Considerations	143
2.	Interrupt Level and Frequency Assignment	143
3.	Selection of Available Interface	145
4.	Interrupt Service Routine	146
5.	Interrupt Handler Installation	146
C.	DYNAMIC MEMORY MANAGER DESIGN	149

1.	IV-SPARC 33 Memory Organization Feature	150
2.	Memory Block for Dynamic Allocation	150
3.	Memory Block Partition	151
4.	Dynamic Memory Allocation Functions Design	152
IX.	MML-11 LANGUAGE SPECIFICATION	153
A.	OVERVIEW	153
B.	DATA STRUCTURE	153
1.	Point	153
2.	Velocity	154
3.	Configuration	154
4.	Path Element	154
C.	USER FUNCTION SPECIFICATION	155
1.	Geometric Functions	155
2.	Motion Planning Functions	158
3.	Motion Control Sequential Functions	159
4.	Motion Control Immediate Functions	163
5.	Sonar Control Functions	167
X.	SENSOR-BASED MOTION NAVIGATION	171
A.	SENSOR-BASED MOTION CONTROL	171
B.	CLEARANCE DEFINITION	172
C.	EDGE-FOLLOWING MOTION	173
D.	VERTEX-FOLLOWING MOTION	174
E.	WALL-FOLLOWING MOTION CONTROL	175
F.	ALGORITHM	178
G.	IMPLEMENTATION	179
1.	Simulation	179

2.	Experimental Results of Real Time System	181
XI.	IMPLEMENTATION	183
A.	OVERVIEW	183
B.	DATA STRUCTURES	183
1.	Data Structure for End-portion Motion Planning	183
2.	Data Structure for Mid-portion Motion Planning	185
C.	LOCAL MOTION PLANNING ALGORITHMS	187
D.	EXPERIMENT RESULTS	187
XII.	CONCLUSIONS	191
APPENDIX A.	FURTHER UNDERSTANDING OF STEERING FUNCTION	193
A.	LIMITATION OF SMOOTHNESS ON LINE TRACKING	193
B.	TWO-WAY LINE TRACKING	198
APPENDIX B.	DYNAMIC MEMORY MANAGEMENT STRATEGY	203
A.	INTRODUCTION	203
B.	BUDDY SYSTEMS	205
APPENDIX C.	USER PROGRAM EXAMPLES	211
LIST OF REFERENCES	217
BIBLIOGRAPHY	219
INITIAL DISTRIBUTION LIST	221

ACKNOWLEDGMENTS

For four years, I have devoted myself to seeking the truth in the sea of knowledge in Naval Postgraduate School. If I was successful in achieving the goal, my own effort was not the only factor. There were many people behind this endeavor. They deserve my appreciation.

First of all, I would like to present my gratitude to my wife Kuei-Lan. She went through and shared all my frustration and excitement during these years. If some one had asked her, "It's five a.m., do you know where your husband is?" She would answer that he is definitely in the laboratory. Without her understanding, encouragement and unconditional support, I wouldn't have accomplished this work.

Second, I want to express my appreciation from my deepest heart to Professor Yutaka Kanayama. His intelligence, patience and guidance made this work possible. With his support and help, I achieved the highest goal in my whole life.

Third, I want to say thank you to the other members of my committee for their enthusiasm and guidance. Professor Mukherjee, Professor Rasmussen, Professor Shing, and Professor Wu provided many valuable comments on the scientific approaches in the motion planning.

Last, I also would like to thank the members of the Yamabico research group in Computer Science Department of NPS for their positive attitude toward my presentations. This gave me lots of courage in preparing for my defense. Best wishes to all.

I. INTRODUCTION

Imagine an autonomous robot vehicle moving indoors. The mission is to provide for its navigation. What are the problems that needs to be solved to accomplish its mission and how can those problems be solved? These have been major sources of research topics in robotics. We refer to these problems as motion planning problems.

The basic motion planning problem was simplified and defined by Jean C. Latombe as follows:

Let \mathcal{A} be a single rigid object -- the robot -- moving in a Euclidean space \mathcal{W} , called *workspace*, represented as \mathcal{R}^n , where $n = 2$ or 3 .

Let $\mathcal{B}_1, \dots, \mathcal{B}_q$ be fixed rigid objects distributed in \mathcal{W} . The \mathcal{B}_i 's are called *obstacles*.

Assume that both the geometric descriptions of $\mathcal{A}, \mathcal{B}_1, \dots, \mathcal{B}_q$ and the locations of the \mathcal{B}_i 's in \mathcal{W} are accurately known. Assume further that no kinematic constraints limit the motion of \mathcal{A} .

The problem: Given an initial position and orientation and a goal position and orientation of \mathcal{A} in \mathcal{W} , generate a path, specifying a continuous sequence of positions and orientations of \mathcal{A} , avoiding contact with the \mathcal{B}_i 's, starting at the initial position and orientation, and terminating at the goal position and orientation. Report failure if no such path exists.[1]

This definition was, of course, oversimplified. However, it stated the essential idea of motion planning. A formal statement of the problem, known as the *configuration space formulation*, then was presented in the book [1 p.7- 11] where the terms, *configuration*, *configuration space* (*Cspace*), *obstacle*, *configuration obstacle* (*Cobstacle*), *free space*, *path*, and *free path*, are defined more precisely.

A. REVIEW OF OTHER MOTION PLANNING APPROACHES

Many different methods have been developed for motion planning over the years. Some of them are applicable to a wide variety of motion planning problem, whereas others have limited applicability. Latombe summarized the general approaches to the motion planning problem into three categories: *roadmap*, *cell decomposition*, and *potential field* [1]. We will briefly introduce these approaches and summarize them below.

1. Roadmap

Roadmap approaches attempt to capture the connectivity of the set of free configurations of a robot in the form of a network of one-dimensional curves called *roadmap* [2]. Once a roadmap is constructed, it is used as a set of standardized paths. Motion planning is thus reduced to connecting the initial and goal configurations to the roadmap and searching the network for a path between these two configurations.

Various methods based on this general idea have been proposed. The best-known types of roadmaps are *visibility graph*, *Voronoi diagram*, *freeway net* and *silhouette*. The *visibility graph* is one of the earliest path planning methods. In two-dimensional Cspace, this approach consists of considering all the vertices of the Cobstacles and the start and goal configurations as the roadmap nodes, while the links are all the line segments connecting two nodes that do not intersect any of the Cobstacles. The visibility graph can be searched for the shortest path between start and goal configurations.

The *Voronoi diagram* is defined as the set of points that are equidistant from two or more objects [3]. The start and goal configurations are retracted in the diagram. A path is searched in the diagram between start and goal configurations. A comprehensive survey of the Voronoi diagram is presented in Aurenhammer [4]. The advantage of this diagram is that it yields free paths which tend to maximize the clearance between the robot and the obstacles.

The *freeway net* method generates paths along generalized cones between the obstacles [5]. The *silhouette* method uses techniques from differential topology. It projects

an object in a higher-dimensional space to a lower-dimensional space and then traces out the boundary curves of the projection, which is called *silhouette*. The silhouette curves are recursively projected to lower-dimensional space, until they become one-dimensional lines. Then the curves are connected at places where new silhouette curves appear or disappear using linking curves. This method is developed to find a path from a graph of one-dimensional curves. It is mostly used in theoretical algorithms analyzing complexity, rather than in developing practical algorithms [6].

2. Cell Decomposition

The *cell decomposition* approaches have been very widely used and are based on decomposing (either exactly or approximately) the set of free configurations into simple non-overlapping regions called *cells*. The adjacencies of these cells are then represented in a connectivity graph. A collision-free path between the start and the goal configuration of the robot is found by first identifying the two cells containing the start and goal configurations and then connecting them with a sequence of connected cells.

Cells can be object-dependent or object-independent. In object-dependent decomposition, boundaries of obstacles are used to generate the cell boundaries, and the union of free cells exactly defines the free space. The number of cells is small, but the complexity of decomposition is high [3] [7]. In an object-independent decomposition, the Cspace is prepartitioned into cells of a simple shape, and each cell is tested to determine whether it is inside or outside of Cobstacles [8] [9].

3. Potential Field

The *potential field* methods generally employ repulsive potential fields around obstacles and an attractive field around the goal. The gradient of the combined potential guides the path away from the obstacles and toward the goal. A historical review of the potential field approach can be found in [10]. The major drawback with these methods is that the potential function will often lead the path to some local minimum from which it cannot escape and therefore causes the planner to fail [11].

4. Layered Approach

Mobile robots are likely based upon carts with wheels for steering and locomotion [12]. It is not possible to follow an arbitrary path. Many constraints need to be taken into account in planning paths for mobile robots. Based on the motion planning categories stated above, numerous algorithms have been developed for autonomous robots over a period of years. Laumond extended the basic motion planning problem to the case of a point robot with kinematic constraints [13]. He developed a method to break down the planning problem into two phases. In the first phase, the problem is solved by finding a collision-free path while ignoring the orientations of robot's start and goal configurations. Then in second phase the path is transformed into a topologically equivalent collision-free path using arcs and tangent line segments. The number of reversals in the path is not limited and the path involving reversals is not smooth.

Another approach to this problem breaks up the free space into convex cells, and computes intermediate configurations at the border of every adjacent pair of cells [14]. Because of the characteristics of convex cells, a simple procedure is devised to connect the intermediate configurations with arcs and tangent lines. Reverse motions are allowed throughout the path in this research. Simulation results showed that the solution path involves many unnecessary reverse motions even in a spacious environment. Moreover, the motions for the initial and final stages were not described clearly.

One of the closely related research is to develop algorithms for motion planning with different layers [5]. This method proposed using Spine net to construct a global path. Then straight line segments and circular arcs were used to plan the detailed motion. The Cubic spiral was adopted to smooth the local motion. This method looks complete but is hard to implement. The other global motion planning and local motion planning ideas can be found in other research reports. Some of these focus on motion planning for manipulators [11]. Some of them provide general ideas [3] [15] [9].

The most recent research in layered motion planning is presented by Kovalchik, J. [7] in our research group. The method used in his research is to first decompose the free

space into K-regions. After this a connectivity graph is built and searched to find a global path from the region that contains the start configuration to the region containing the goal configuration. A bidirectional motion planning method is also introduced in this research. In short, this research concentrates on the global path planning which will be adopted in this dissertation for the top layer of the autonomous vehicle motion planning.

B. SCOPE OF DISSERTATION

This dissertation solves the motion planning problem under the assumption to be described in Section C of this chapter. Our focus is on planning the motion as a physical approach. The algorithms for motion planning to be taken by real robot are deliberately developed. Simulations are done to verify the algorithms either individually or globally.

After this a high-level motion specification language, Model-based Mobile-robot Language (MML-11), which is suitable to describe the solutions to the motion planning problem, is developed as the implementation and verification of the algorithms. The dissertation also includes the description of MML-11.

C. MOTION PLANNING PROBLEM

1. Problem Statement

The world space \mathcal{W} for the motion planning problem in this dissertation is a two-dimensional plane \mathbb{R}^2 on which a global Cartesian coordinate system is defined. The obstacles are closed subsets of \mathcal{W} . In this dissertation obstacles are assumed to be simple, rectilinear, and polygonal. The free space, $Free(\mathcal{W})$, is the complement of the union of all obstacles in the world space \mathcal{W} .

The vehicle in the dissertation refers to a rigid-body robot which has fixed size. (During the research we will use Yamabico-11 for experiments although the algorithms will be suitable for robots with any size).

A **configuration** q in this dissertation is defined as a triple $q = (p, \theta, k)$, where p indicates the position (x, y) in the global Cartesian coordinate system, θ is the orientation

related to the x-axis of the global coordinate system [16], and k is the specified curvature. The configuration defined in this dissertation is normally used to describe the robot's instantaneous status, either it is stationary or moving. This configuration is especially useful for specifying a path. For instance, if we use $q = ((x, y), \theta, k)$ to specify a line, this line passes through the point at (x, y) and with orientation θ . When the curvature element is $k = 0$, it is specifying a straight line, otherwise it is a curve.

The motion of a vehicle is subject to nonholonomic and kinematic constraints. That is, the vehicle is able to perform both forward and backward motion but not sideways motion. The robot's orientation and curvature are continuous. There is an upper limit in its curvature when a turn is taken.

The problem to be solved in this dissertation is: In a given world \mathcal{W} , with free space $Free(\mathcal{W})$, let q_s and q_g be the start and goal configurations, respectively, lying completely inside free space. Let \mathcal{A} be the wheeled mobile robot vehicle. Given a mission to move from q_s to q_g , this research is to plan a safe motion symmetrically for the vehicle under the given constraints, to achieve its mission.

2. Layered Motion Planning

As stated in Section A, numerous methods have been developed for motion planning. These methods are variations of a few general approaches: road map, cell decomposition, potential field, and mathematical programming [3] [1]. Some of them are applicable to a wide variety of motion planning problems, whereas others have a limited applicability. Unfortunately, none of them are complete in the sense of practical use for solving the motion planning problem defined in this dissertation. For example, nonholonomic constraints and kinematic constraints were not taken into consideration in many approaches. In particular, the robot's motions in the area of the start or goal configurations are more restricted, and so require more deliberative planning. Not all robotics systems proposed for motion planning are developed to address this consideration. Furthermore, most of research in motion planning for mobile robot is theoretically valuable

but not practically useful. For these reasons, we propose a new approach which divides the motion planning into two layers: the **global path planning** and the **local motion planning**, as shown in Figure 1.1.

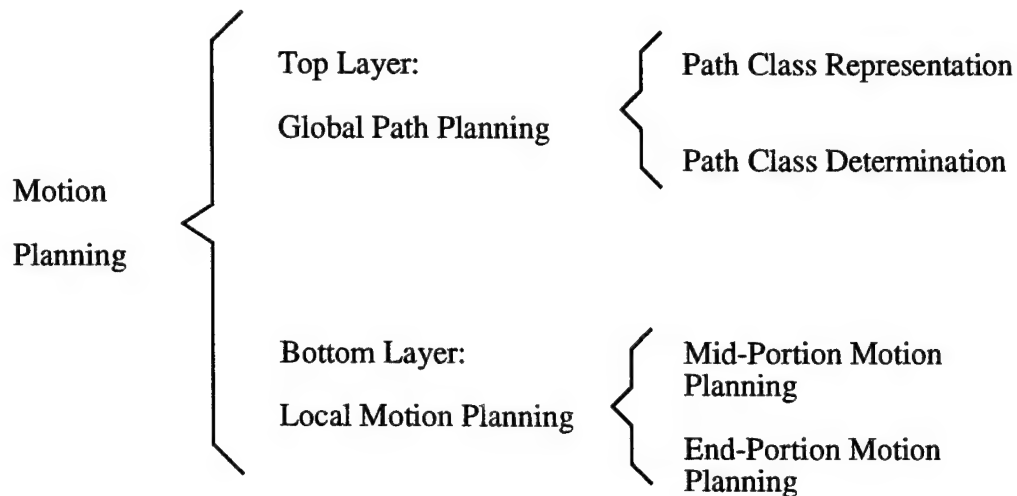


Figure 1.1: Layered Structure of Motion Planning

The top layer, global path planning, starts with decomposition of the free space of the given world into K-regions. Then a connectivity graph is built and searched to determine path classes which classify equivalent paths. At completion of motion planning in this layer, one path class among all distinct classes is determined. This path class, represented by a sequence of alternative region-borders called a crossing sequence [7], specifies the direction of a possible path for a robot, which will continue to plan its detailed motion in next layer. The details of top layer motion planning will be reviewed in Chapter II.

With the global path determined, the motion planner at the bottom layer then takes the general direction guided by the global path to direct its local motion planning. The task of this layer is to produce a smooth collision-free motion for the robot. We further divide this layer into two types of motion planning: the **mid-portion motion planning** and the

end-portion motion planning. In mid-portion motion planning, the robot's motion will be planned region-by-region along the global path, excluding the regions in or near the first and last regions. The end-portion motion planning determines the motion of starting or ending from start or final configurations which are called **initial motion planning** and **final motion planning** respectively. Both types of local motion planning are done under nonholonomic and kinematic constraints. The steering function and forerunner simulation are powerful methods to find solutions in this layer. The bottom layer motion planning will be discussed in detail in Chapters III, IV and V. The structure of motion planning and motion execution is shown in Figure 1.2.

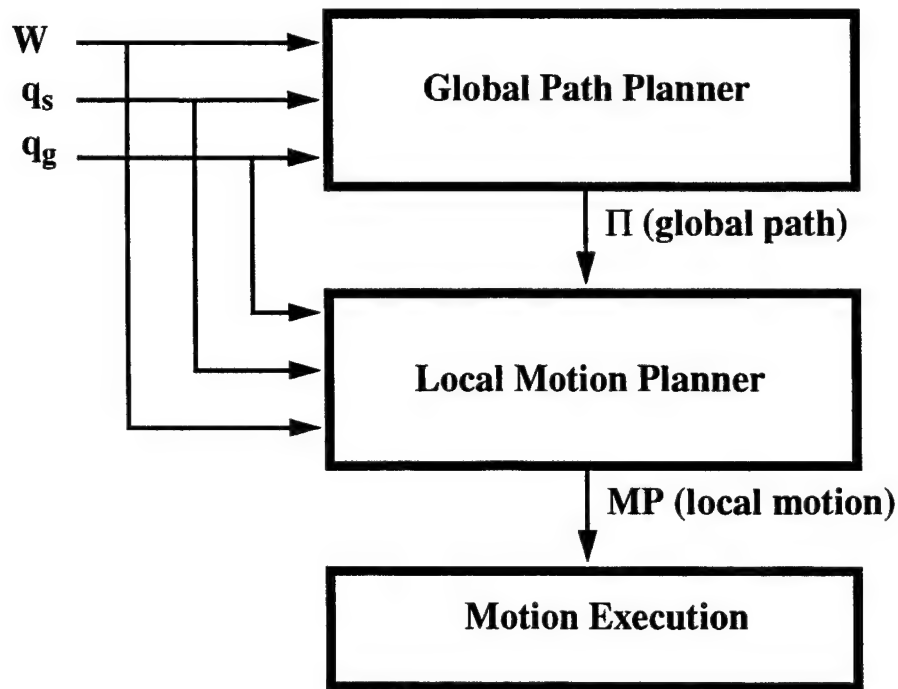


Figure 1.2: Structure of Motion Planning and Execution.

3. Assumptions

This research is based on the following assumptions:

- Two dimensional world: The world space is a flat planar world with obstacles.

The floor is on the x, y plane.

- Rectilinear obstacles: All obstacles in the dissertation refer to rectilinear, polygonal obstacles which have their edges perpendicular to the x, y axis of the coordinate system.
- Stationary environment: The environment in this dissertation refers to a stationary world space.
- Gross motion planning: We assume that free space is much wider than the objects' size with allowance for the positional error of the robot. This ensures that position error will not cause unexpected collisions while executing the collision-free paths generated by the motion planner. The motion planning under this assumption is called **gross motion planning** [3].
- No localization error: In this dissertation, we assume that the localization error has been corrected. Thus the localization error does not exist during the execution of the planned motion.
- The motion is subject to constraints of robot's kinematic and nonholonomic characteristics.
- The robot is a rigid-body autonomous vehicle.

D. ORGANIZATION OF DISSERTATION

This dissertation provides a solution to the motion planning problem for autonomous vehicles. It includes descriptions of existing planning approaches, design of a robotic language, and implementations. In addition, an experimental robot, Yamabico-11, which plays a significant role during the entire project, will be introduced. The dissertation is organized as follows:

Chapter I gives an overview of the dissertation. It reviews the general motion planning approaches, defines the scope of the dissertation, and states the problem to be solved.

Chapter II describes the global path planning. We review the theory of homotopy path classes and the theory of K-decomposition that will be adopted in the top-layer planning. Then a path-finding method for building the global path represented by K-regions is discussed.

In Chapter III, we break down the local motion planning into two types of planning: end-portion motion planning and mid-portion motion planning. We present the analysis of local motion planning tools to be used in this dissertation. The characteristics of steering function are studied in detail. The concepts of Forerunner simulation and reverse path are described. After then, we discuss the steps to be taken in local motion planning.

Chapter IV discusses mid-portion motion planning in detailed. The planning methods for various types of K-region on the path are identified and analyzed. The algorithms for planning the robot's motion in regions other than the initial and final ones are developed.

Chapter V gives a detailed description of end-portion motion planning including the initial motion planning and final motion planning. Planning tools are described prior to the discussion of planing details.

Chapter VI introduces the hardware of the experimental robot Yamabico-11.

Chapter VII, VIII, and IX describes the design of a robotic software system -- Model-based Mobile robot Language, MML-11. Chapter VIII states the design of robot real time operating system. In Chapter IX, the language specifications and software architecture are described.

Chapter X presents a sensor-based approach to motion control

Chapter XI and XII present results and conclusions.

II. GLOBAL PATH PLANNING

A. HOMOTOPY

We consider a two dimensional world, W , with holes. A **hole** is an obstacle for a robot. A free space $Free(W)$ is the complement of the union of all holes. There might be a hole among them which completely surrounds the free space. A hole with this property is said to be **inverted**. Every free space is a connected subset of W (Figure 2.1).

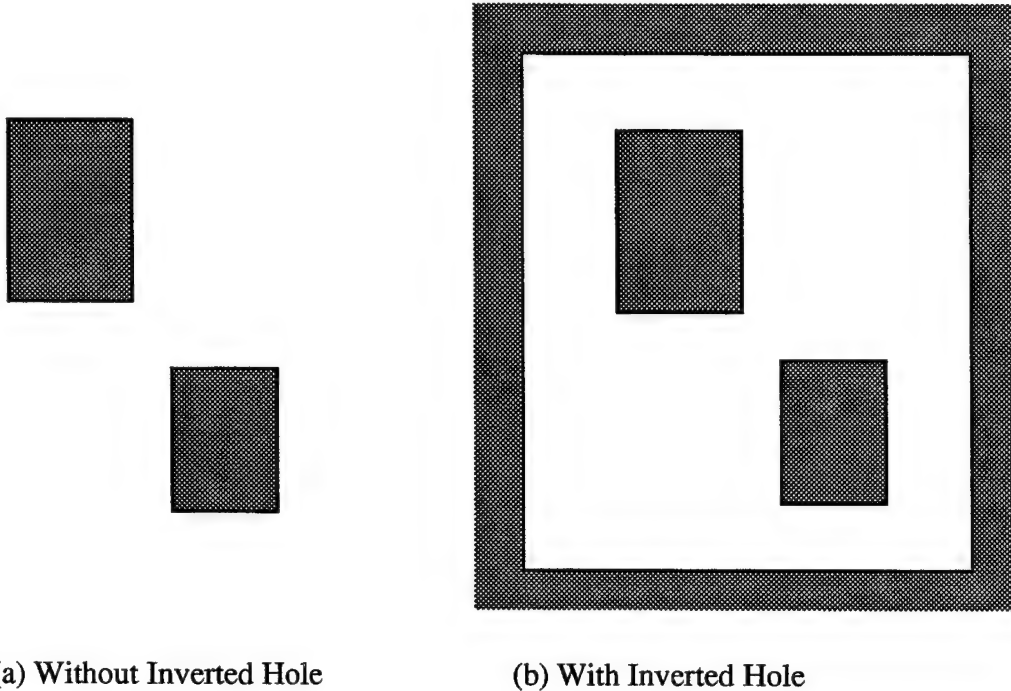


Figure 2.1: Two-Dimensional World Space with Holes

A **directed curve** or **directed path** Π is represented by a continuous function $f: [0, 1] \rightarrow Free(W)$. The two points $f(0)$ and $f(1)$ are called **endpoints** of Π . The path is said to **join** the endpoints. Obviously, there are infinitely many paths joining given two points S and G in $Free(W)$ (Figure 2.2). In this figure, paths Π_1 and Π_2 are somewhat similar and so are paths Π_3 and Π_4 . However, Π_1 and Π_3 are not. This concept was formally defined in

the field of algebraic topology [17] and will be followed. Two paths Π and Π' (defined by f and f' respectively) are said to be **homotopic** if and only if there exists a continuous function $\phi: [0,1] \times [0,1] \rightarrow \text{Free}(W)$ such that

1. $\phi(0, t) = f(0)$ for all $t \in [0, 1]$,
2. $\phi(1, t) = f'(1)$ for all $t \in [0, 1]$,
3. $\phi(s, 0) = f(s)$ for all $s \in [0, 1]$, and
4. $\phi(s, 1) = f'(s)$ for all $s \in [0, 1]$.

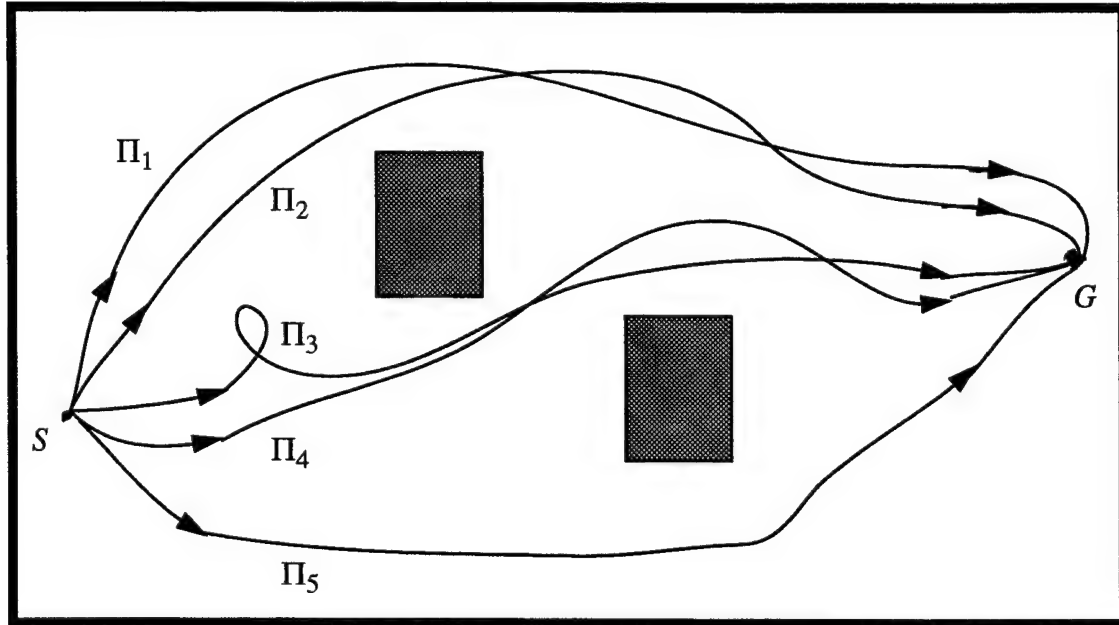


Figure 2.2: Homotopic Paths

Informally speaking, Π_1 can be continuously transformed into Π_2 , without running over any holes, with both endpoints fixed. So are paths Π_3 and Π_4 . If two paths Π and Π' are equivalent, we write

$$\Pi \equiv \Pi' \quad (\text{Eq 2.1})$$

Therefore in Figure 2.2, $\Pi_1 \equiv \Pi_2$ and $\Pi_3 \equiv \Pi_4$. There are a countable number of equivalence classes of paths, even in a world with a finite number of holes.

B. OVERVIEW OF CONVEX POLYGONAL K-REGIONS THEORY

The ability to decompose a world into regions which capture the topology of the space in terms of distinct homotopy classes and which provide useful data for the local motion planner is important in global path planning. Joe Kovalchik developed the theory of convex polygonal K-regions in his dissertation [7]. We will adopt this theory to carry out the global path planning for the solution to the top layer of entire layered motion planning. This section reviews the theory briefly and leave the details to [7].

1. K-decomposition

The K-decomposition produces convex regions which are easier to handle in the local motion planner. The global plan, called the **path class**, is a sequence of convex K-regions, which in turn specifies a sequence of borders belonging to these K-regions which must be crossed in order to execute the global plan.

a. Borders

Consider the problem of symbolically representing each homotopy class. A method based on “borders” is presented. (See Figure 2.3) A **border** in a world is a closed line segment B which satisfies the following conditions:

- (i) Both endpoints are on the world’s physical boundary, ∂W , and
- (ii) The open segment of B is a subset of $Free(W)$.

b. Decompositions and Regions

In a world, a finite set

$$\mathcal{L} = \{B_1, \dots, B_m\} \quad (\text{Eq 2.2})$$

of borders in the world in which two borders’ open segments do not intersect is called a **decomposition** (of the free space $Free(W)$). Thus \mathcal{L} divides $Free(W)$ into a finite

number of **regions**. A region, R , includes borders to which it abuts as part of its boundaries. Therefore, a border belongs to at most two regions.

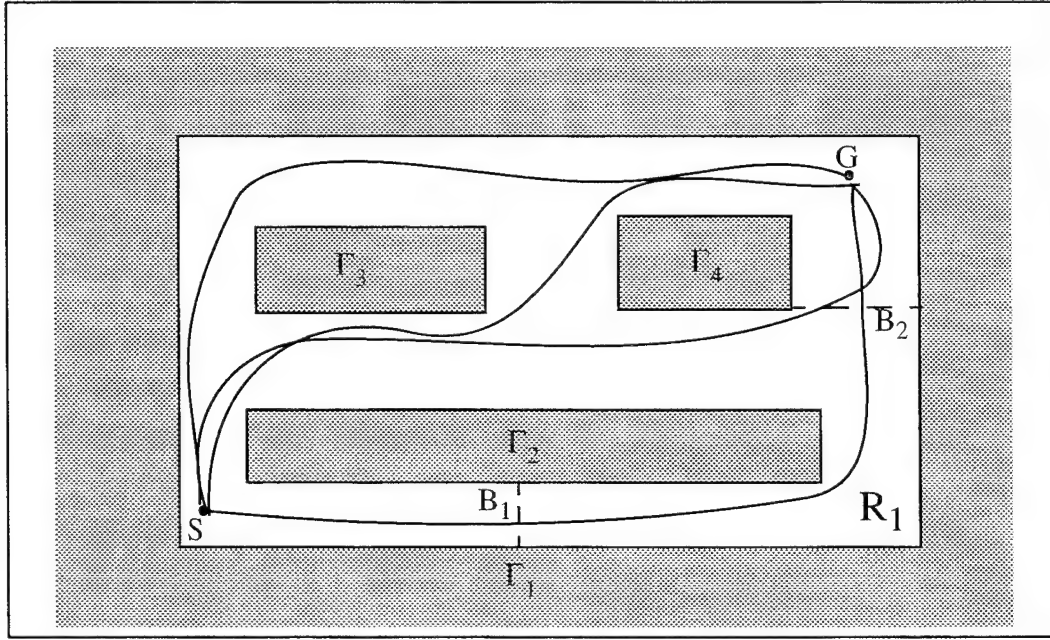


Figure 2.3: Borders

c. Normal Path and Crossing Sequence

A **normal path** is a path which does not travel along a border, osculate a border or turn back when crossing a border. The sequence of borders B_{j_i} , for $i = 1, \dots, k$, is called the **border sequence**. The sequence of regions R_{m_i} , for $i = 0, \dots, k$, is called the **region sequence**. For a normal path, f , the following sequence, $\beta(f)$, is called the **crossing sequence**:

- (i) $\beta(f) = R_{m_0} B_{j_1} R_{m_1} B_{j_2} R_{m_2} \dots B_{j_k} R_{m_k}$, ($k \geq 0$), where
- (ii) the start configuration is located in $R_{m_0} = R_s$
- (iii) the goal configuration is located in $R_{m_k} = R_g$, and

d. Convex K-decomposition

Let $R(W, \mathcal{L})$ be defined as the set of all regions contained in $Free(W)$ created by a decomposition \mathcal{L} in W . When \mathcal{L} is a decomposition of $Free(W)$ such that every region of $R(W, \mathcal{L})$ is convex, it is called a **K-decomposition**. (see Figure 2.4)

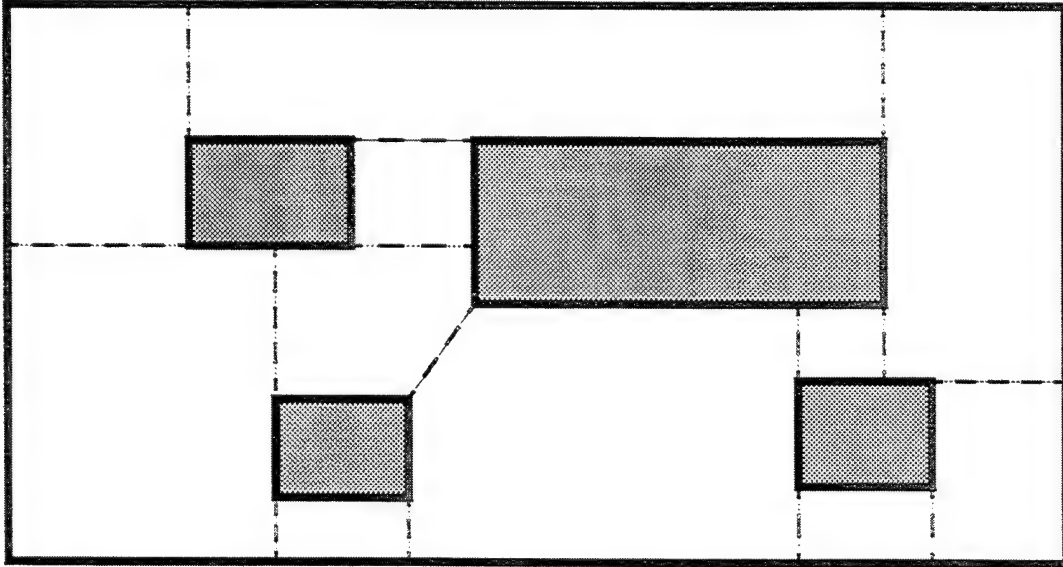


Figure 2.4: The Example of K-decomposition

Obviously, there is more than one K-decomposition for a given W . A convex region so created by the above decomposition is called a **Convex K-region** and satisfies the following conditions: i) the interior of the convex polygon lies completely in free space, ii) each vertex defining the convex polygon is contained in a physical boundary of W .

2. Global Path Planning

Given a world W , start configuration q_s , and goal configuration q_g , the global path planning problem is to find an optimal path connecting the K-region containing q_s with the K-region containing q_g . This planning is done by first decomposing $Free(W)$ into K-

regions, then building a connectivity graph, and finally using a modified Dijkstra's algorithm to search for an optimal path represented by a crossing sequence. The K-decomposition is introduced as described in the previous subsection. In this subsection we review how the optimal path is obtained after K-decomposition of the world is done.

a. Connectivity Graph

Given a K-decomposition of a world, its geometric adjacency relationship is represented by a directed connectivity graph, \mathcal{G} . Each K-region is considered as a node of \mathcal{G} , and each border is considered as an edge in \mathcal{G} . Figure 2.5 is a decomposed world model with border and region names assigned. The connectivity graph corresponding to this world model is shown in Figure 2.6. and 2.7

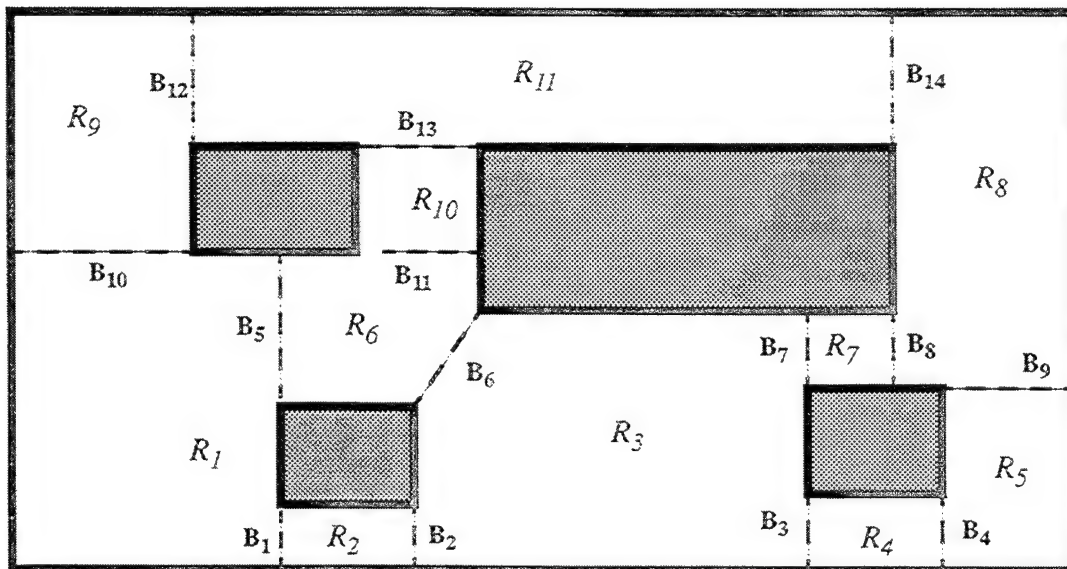


Figure 2.5: A K-decomposition of World Model

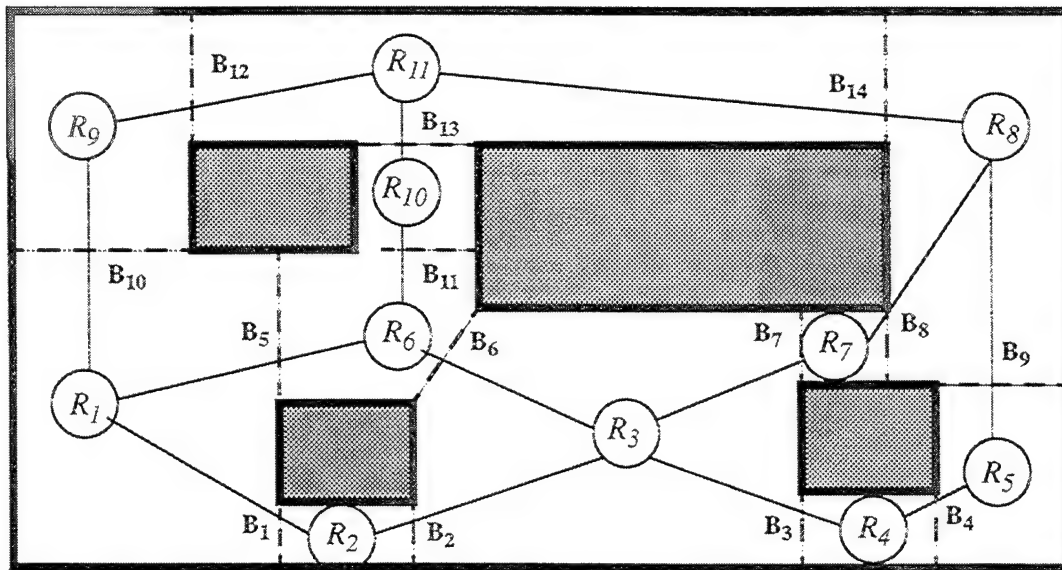


Figure 2.6: The Connectivity Graph of a K-decomposition of World Model

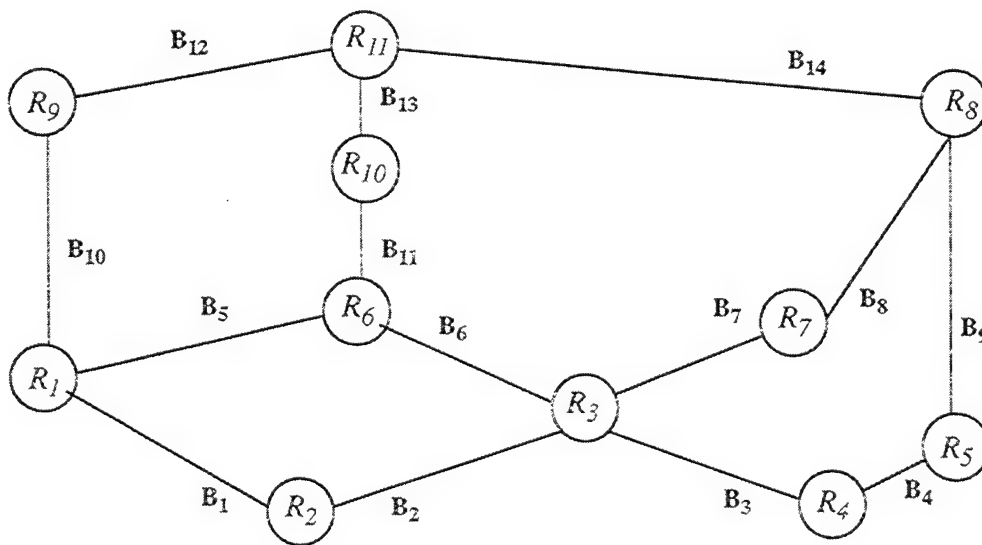


Figure 2.7: A Connectivity Graph

b. Path Class Representation

Consider the problem of finding a path from a start configuration q_s to a goal configuration q_g in a K-decomposed world. In its most general form, a path class, P , is represented by the crossing sequence. In the example of Figure 2.8 and 2.9, one of the path classes, which connect region R_1 with region R_8 , is represented by a crossing sequence as follows:

$$P = \langle R_1, B_5, R_6, B_6, R_3, B_7, R_7, B_8, R_8 \rangle \quad (\text{Eq 2.3})$$

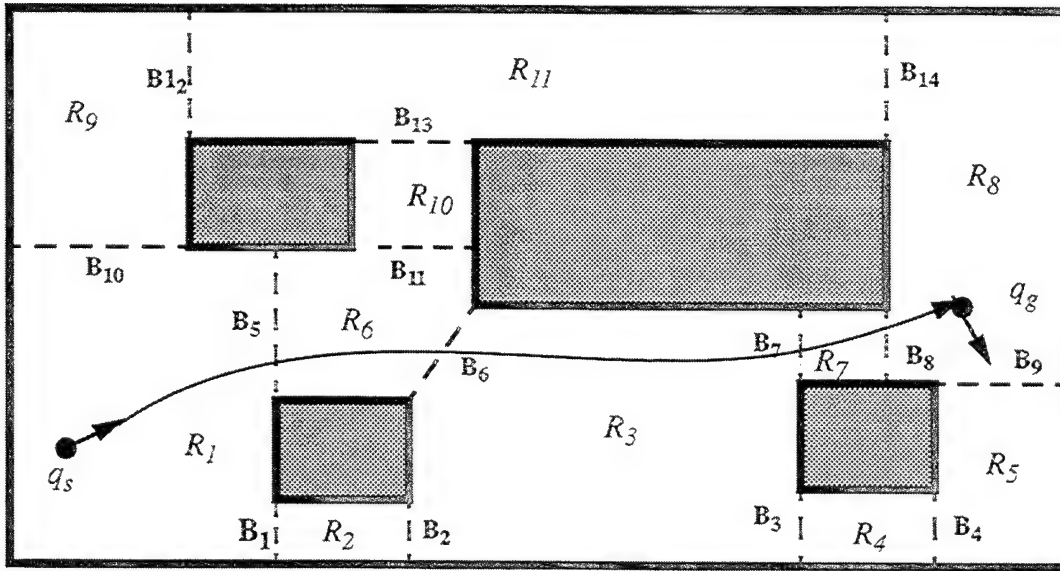


Figure 2.8: A Path Class in the K-decomposed Regions

c. Finding Optimal Path Class

A modified Dijkstra's algorithm is used to find an optimal path class given two configurations, q_s and q_g . The path class will be found in terms of the crossing sequences. Both position and orientation of the start and final configurations are taken into

consideration when performing the search for the minimum cost path class. Relaxation of edges involving the start and the goal regions determines a cost by using a bidirectional motion-planning algorithm. This algorithm results in achieving a better approximation of the total cost of the path class by considering the maneuvers conducted in the initial and final motions.

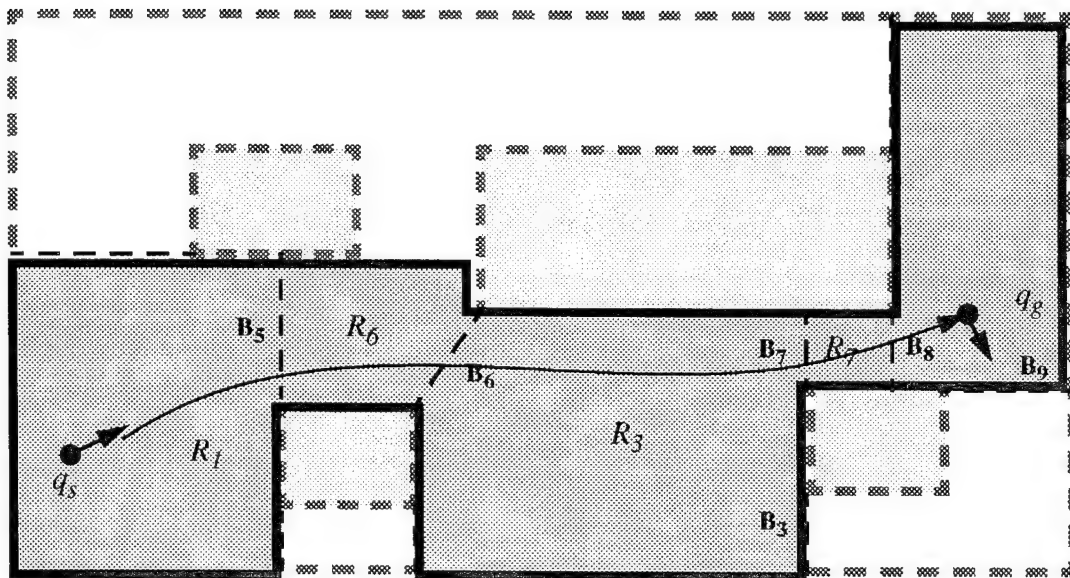


Figure 2.9: Highlight of the Path Class in Figure 2.8

III. LOCAL MOTION PLANNING APPROACH

The regions on the global path class provide information for rough robot navigation. A safe motion plan for accomplishing the mission will not be ensured without further elaborative planning. The task of local motion planning is to produce a smooth, collision-free motion for the robot based on the global path class generated by global path planner. This chapter addresses an approach to the local motion planning. This approach provides the fundamental concepts to be used in the local motion planning of this dissertation. In Section A, we state the local motion planning problem. Section B describes the preliminary planning decision. In Section C, we will introduce a solution to the mobile robot motion control -- steering function. Section D discusses one of planning methods -- forerunner simulation. In Section E, the concepts of symmetric path and reverse path are presented. Section F describes the local motion planning steps. The planning details will be further discussed in Chapter IV and Chapter V.

A. PROBLEM STATEMENT

The robot's motion is said to be *safe* if its trajectory is collision-free. The motion planning starting from configuration q_1 to configuration q_2 is said to be *symmetric* if the trajectory of the planned motion is exactly the same as the trajectory of the motion planned reversely (from q_2 to q_1 with reverse orientations). The local motion planning is as follows:

Given a K-decomposed world model W , a start configuration q_s , a goal configuration q_g , and a global path class $\Pi = \langle R_{m_0} B_{j_1} R_{m_1} B_{j_2} R_{m_2} \dots B_{j_k} R_{m_k} \rangle, k \geq 0$, in which q_s locates in region R_{m_0} and q_g in region R_{m_k} . The problem of local motion planning is that of planning a safe motion symmetrically for a rigid body robot to travel from q_s to q_g with the global path class.

For example, a given world is decomposed into K-regions as Figure 3.1. The start configuration q_s resides in R_1 , and goal configuration q_g in R_8 . The given global path class is $\Pi = \langle R_1, B_5, R_6, B_6, R_3, B_7, R_7, B_8, R_8 \rangle$, where the regions (covered by light shaded areas)

and borders between them are as depicted in Figure 3.1. The task for local motion planner is to plan a safe motion for the robot. Its outputs will be a safe and symmetric motion plan. The detail will be discussed in Chapter IV and V.

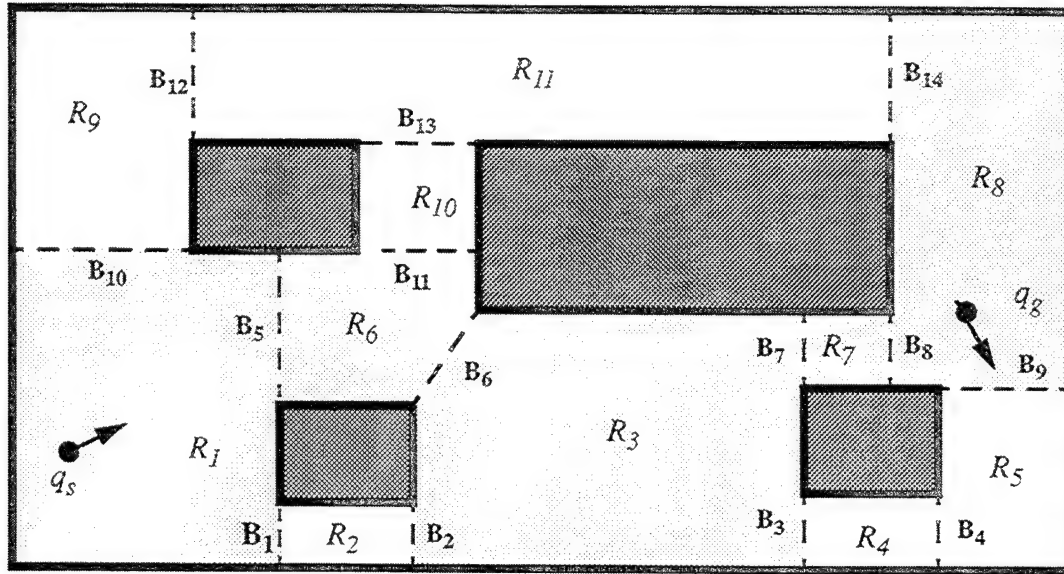


Figure 3.1: The Example of K-decomposition

B. PLANNING APPROACHES

1. Break Down Planning

The global path class is the input to local motion planning. It provides useful information in directing the robot to accomplish its mission. That information is contained in K-regions separated by borders. It is convenient to break down the entire motion planning into individual regions. Therefore, the first decision of local motion planning is made to carry out the planning region-by-region along the global path class.

With nonholonomic and kinematic constraints, the robot's motions near the start configuration and goal configuration should be planned separately in order to achieve the entire motion-planning task. Therefore, the local motion planning is further subdivided into two types of motion planning:

- end-portion motion planning and
- mid-portion motion planning

The **end-portion motion planning** deals with the motion taking place in the regions near two ends of the global path class as shown in Figure 3.2, and the **mid-portion motion planning** plans the robot's motion exclusively between the regions near the two ends as illustrated in Figure 3.3. As stated in Chapter II, the **initial region** is the first region on the global path class, in which the start configuration resides. The **final region** is the last region on the global path class, in which the goal configuration falls. Ideally, the end-portion motion planning involves only initial region and final region. However, in some cases, the regions next to the initial region and final region may be included in end-portion motion planning. The end-portion motion planning which involves the initial region is called **initial motion planning**, and the planning which involves the final region is called **final motion planning**. This will be discussed in Chapter V.

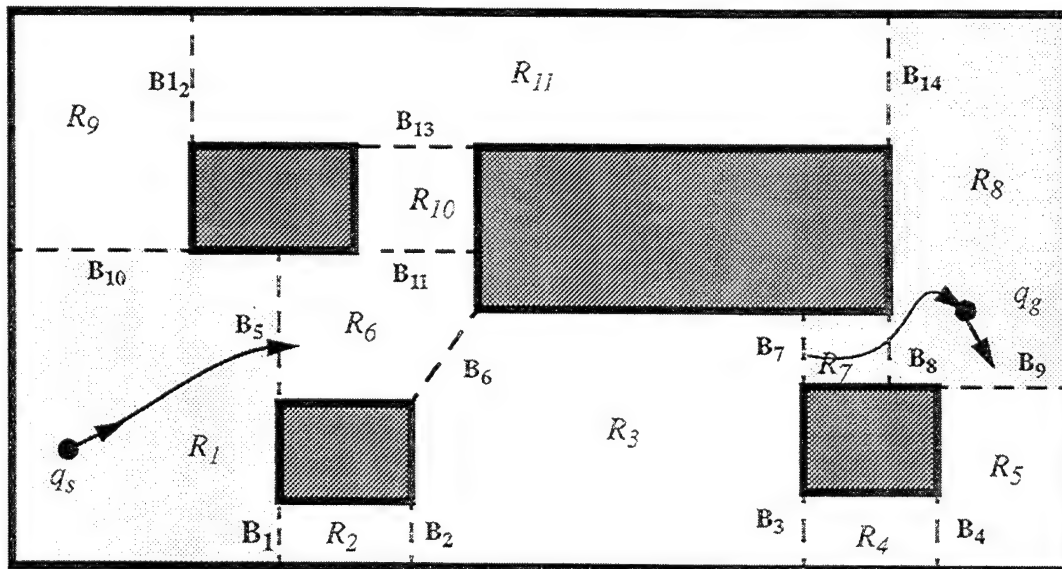


Figure 3.2: The End-portion Motion Planning

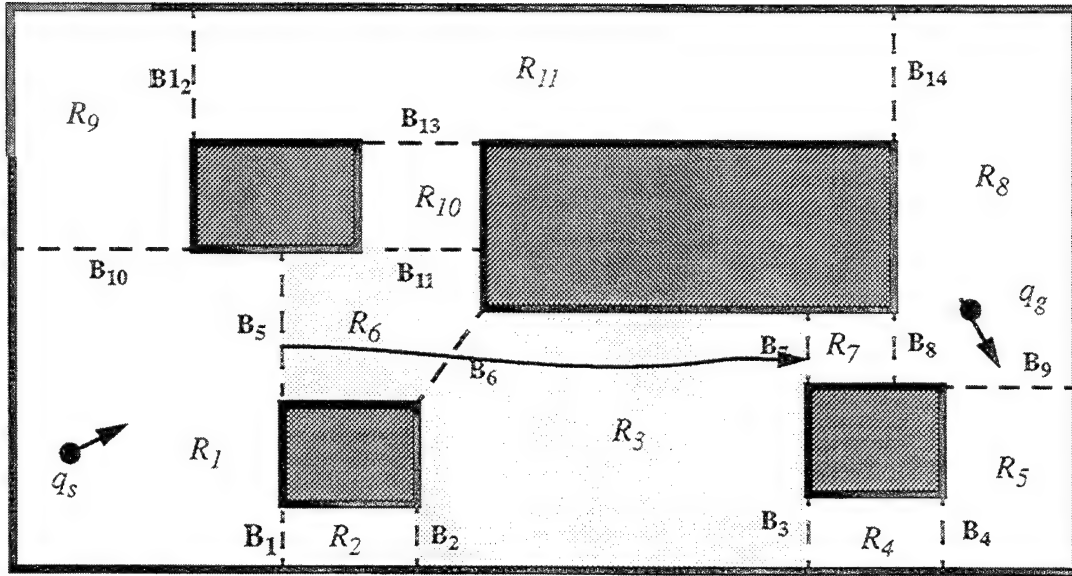


Figure 3.3: The Mid-portion Motion Planning

2. Crossing Border with Orthogonal Orientation

Under the assumption in Chapter I, obstacles are assumed to be rectilinear polygons. Thus, when the free space is decomposed into K-regions, it is natural that the K-regions inherit rectilinear features even though there are a few exceptions. Since the orientation of borders is orthogonal in most cases, planning a motion that crosses a border at the center or at the point with minimum clearance from objects, with orthogonal orientation will be considered safe. This will be the essential idea of local motion planning in the dissertation. With a predetermined crossing point, the border's orthogonal orientation can define a border configuration. Two such configurations on the distinct border of a region can be taken to plan a safe motion in the region. Thus, our primary strategy for local motion planning is to cross borders at crossing points with orthogonal orientation (see Figure 3.4). Using border configurations in local motion planning links end-portion and mid-portion motion planning together into a complete motion plan. When the entire motion plan is finished, it will be as shown in Figure 3.5.

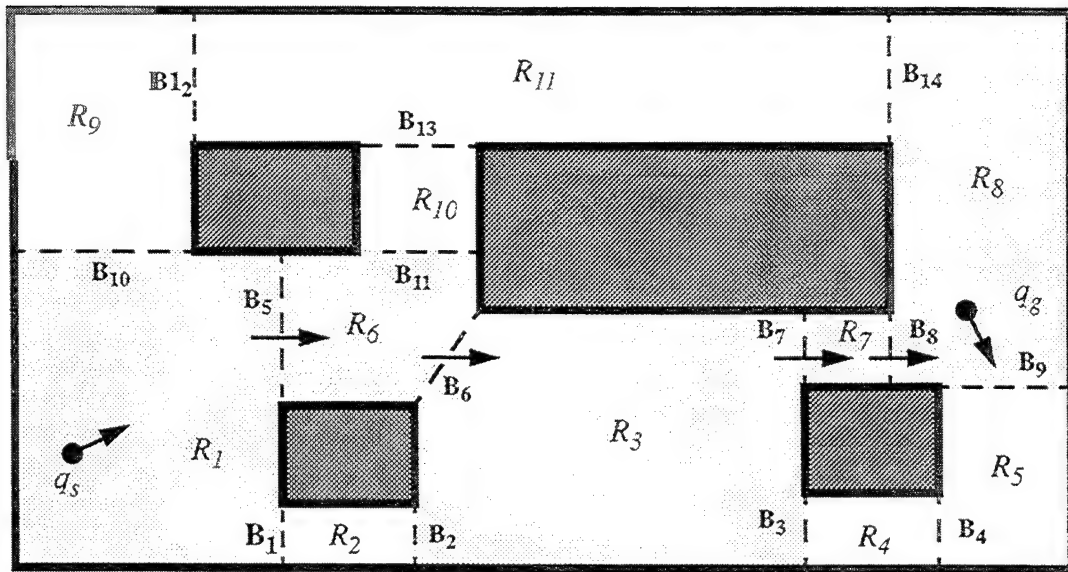


Figure 3.4: Crossing Borders with Orthogonal Orientations

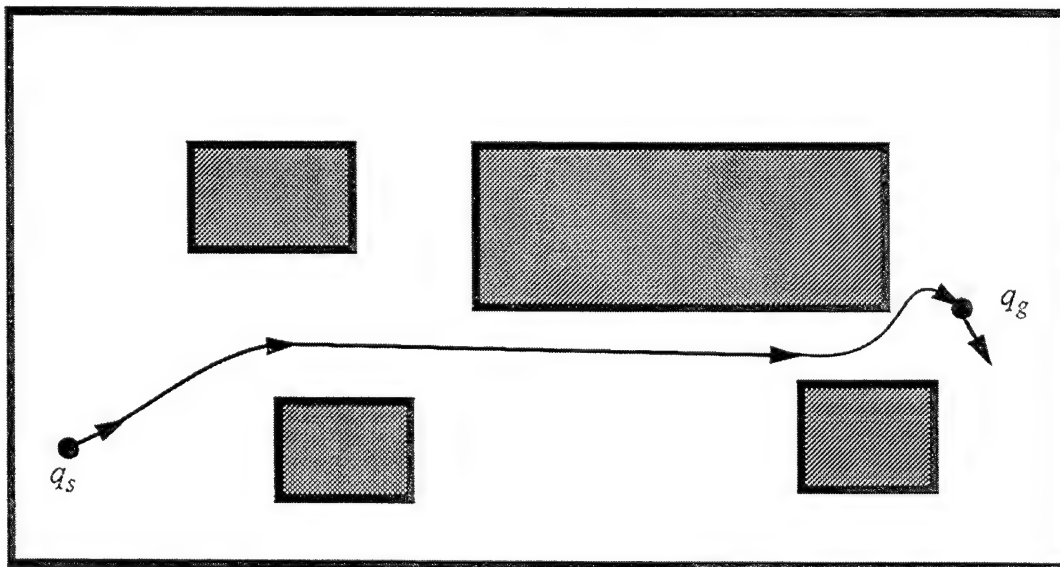


Figure 3.5: A Complete Motion Plan

C. TERMINOLOGY

A few other terminologies and concepts which are useful for the discussing the local motion planning are defined in this section. A K-region is a convex polygon [7]. The boundary of a K-region consists of a set of line segments. Each line segment is an edge of the K-region. An edge of a K-region can be a border, a portion of physical boundary or the combination of these two. The region in which the start configuration falls is called the **initial region**. The region in which the goal configuration falls is called the **final region**.

DEFINITION 3.1 *If a border is one of the edges of a K-region, the border is called a **Full-border** or **F-border**. Otherwise it is called a **Partial-border** or **P-border**.*

DEFINITION 3.2 *Let $C = \langle R_1, \dots, R_n \rangle$ be region sequence of a global path class, where $n \geq 2$. For any region R_i , $2 \leq i \leq n$, there must exist one and only one border between region R_{i-1} and R_i . This border, denoted as B_{in} , is called the **entrance border** or **Entrance** to the region R_i . Similarly, the border between region R_i and R_{i+1} , $1 \leq i \leq n$ is called the **exit border** or **Exit** to the region R_i , denoted as B_{out} .*

DEFINITION 3.3 *Let $v_1=(x_1, y_1)$, $v_2=(x_2, y_2)$ be two end points of a border in a K-region. The orientation of the border is defined as an orientation which is perpendicular to $\arctan2(y_1 - y_2, x_1 - x_2)$. The orientation of the entrance border is denoted by Ψ_{in} and the orientation of the exit border is denoted by Ψ_{out} .*

For a specific border (either the entrance border or the exit border in a region of a path), the orientation of the border is decided by the direction of crossing border motion. For instance, let $v_1=(x_1, y_1)$, $v_2=(x_2, y_2)$ be two end points of the entrance border. If v_1 is on the left side of the entrance direction, then the orientation of the border is computed as $\Psi = \arctan2(y_1 - y_2, x_1 - x_2) - \pi/2$. Figure 3.6 and Figure 3.7 illustrate these concepts in DEFINITION 3.1, DEFINITION 3.2 and DEFINITION 3.3.

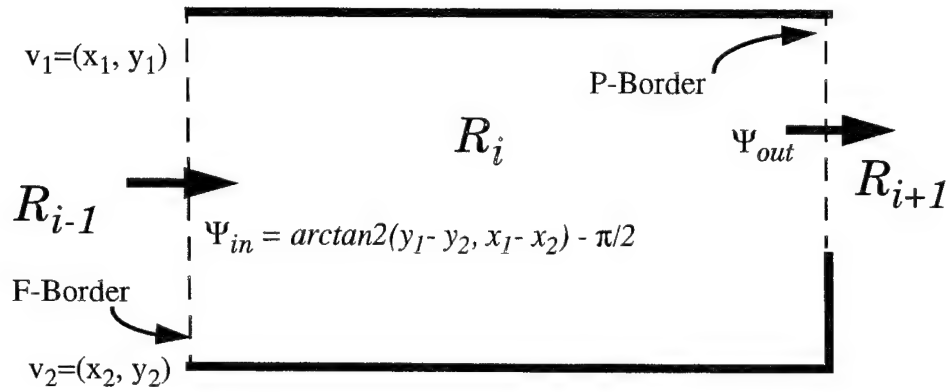


Figure 3.6: Full-Border, Partial-Border and Orientation of Border.

For a region in a global path class, the edge that contains the entrance border is called the **Entrance Edge**, denoted by E_{in} , and the edge that contains the exit border is called the **Exit Edge**, noted by E_{out} . We also denote the center of an entrance edge as EC_{in} , the center of an exit edge as EC_{out} , the center of an entrance border as BC_{in} , and the center of an exit border as BC_{out} . For any rectangular region, there are two pairs of edges. The pair of edges that are parallel to the orientation of the entrance border is called **Forward Edge**, denoted by FE . The length of Forward Edge is called **Forward Length** of the region, denoted by FL . The pair of edges which is perpendicular to the entrance border is called **Cross Edge**, denoted by CE . The length of Cross Edge is called **Cross Length**, denoted by CL . Figure 3.7 shows the definitions. A configuration defined by a point P_{in} on the entrance border, the orientation Ψ_{in} and a curvature k_{in} (normally zero) is called an **entrance configuration**, denoted by $q_{in} = (P_{in}, \Psi_{in}, k_{in})$. A configuration defined by a point P_{out} on the exit border, the orientation Ψ_{out} , and a curvature k_{out} (normally zero) is called an **exit configuration**, denoted by $q_{out} = (P_{out}, \Psi_{out}, k_{out})$. When the curvature is set to 0, the entrance configuration and exit configuration specify a straight line that passes the entrance and exit border respectively.

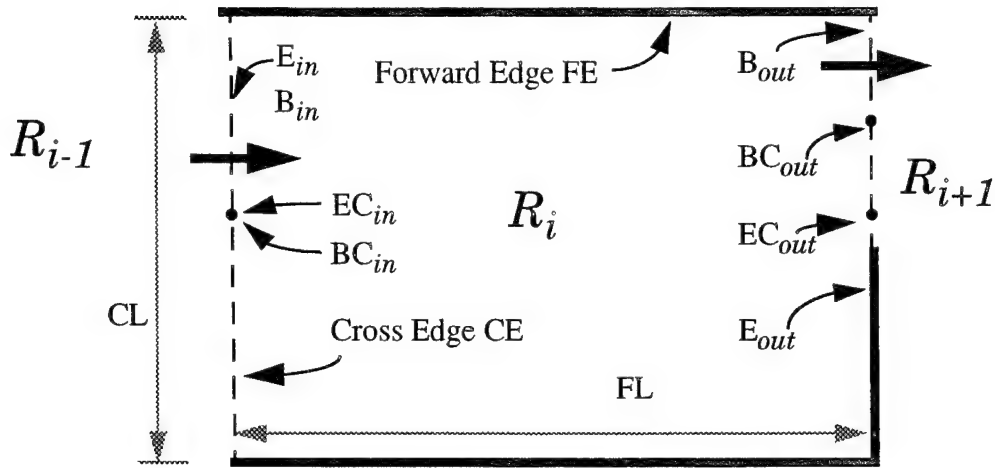


Figure 3.7: Entrance Border, Exit Border, Entrance Edge, Exit Edge and their Centers.

Basically, there are three major ways for entrance and exit borders to be positioned on the edges of a K-region. We categorizes intra-region motions in a K-region into following three types:

- Type I: The entrance and exit borders are parallel. (Figure 3.8)
- Type II: The entrance and exit borders are on edges which share the same vertex of a K-region. (As a special case, the entrance and exit share the same vertex of the K-region.) (Figure 3.9)
- Type III: The entrance and exit borders are on the same edge of the K-region (Figure 3.10)

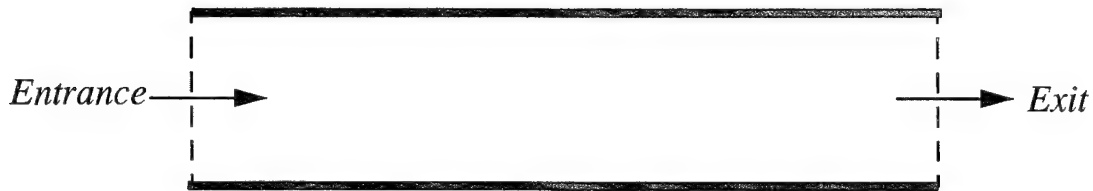


Figure 3.8: An Intra-region Motion of Type I

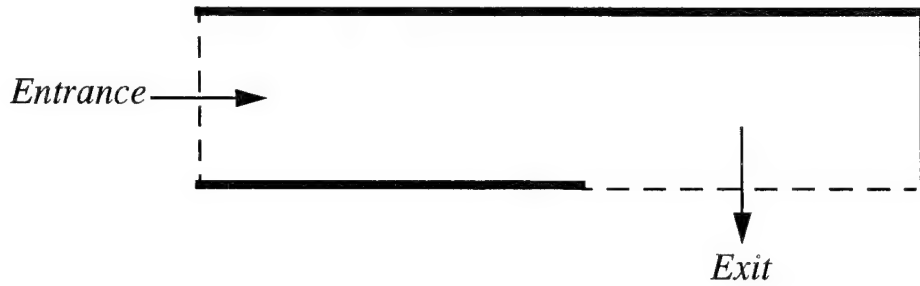


Figure 3.9: An Intra-region Motion of Type II

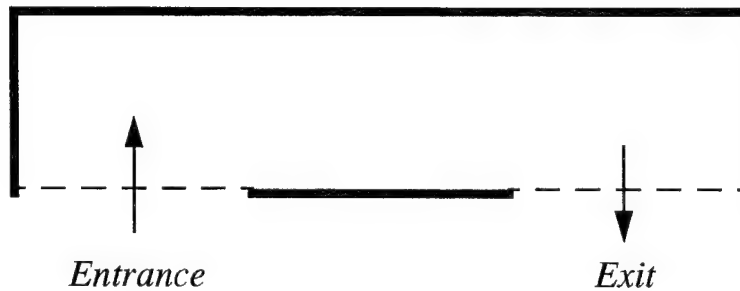


Figure 3.10: An Intra-region Motion of Type III

D. MOTION PLANNING WITH STEERING FUNCTION

1. Smooth Motion Planning with Curvature

An ordinary non-holonomic ground robot vehicle has two control variables: the curvature k of the motion trajectory and the linear speed v . Since a non-holonomic robot's heading orientation θ is always equal to the trajectory's tangent orientation, the vehicle's rotational speed ω is equal to kv (because $\omega = d\theta / dt = (d\theta / ds) (ds/dt) = kv$, where t is time and s is the traveling length of the robot). Therefore, the smooth motion planning for a mobile robot is to design (k, v) or (ω, v) as function of t or s .

This control model with curvature is useful for vehicles with any kinematics [18]. If a vehicle has differential drive type kinematics, with a tread of $2W$, its right wheel speed v_+ and left wheel speed v_- should be $v_{\pm} = (1 \pm W k) v = v \pm W\omega$. If a vehicle has bicycle type kinematics with a distance L between two wheels, the orientation ϕ of the front steering wheel should be $\phi = \arctan(L k)$. Thus the real time evaluation of (k, v) appears to be the most direct method in smooth vehicle control.

One obvious method is to compute the curvature directly as a function of the geometrical constraints and the mission. However, one drawback of this method is that when some of the input has a discontinuity from the previous value, the output k also tends to be discontinuous. As widely known, a rigid body motion with a discontinuous curvature function is not physically realizable. In order to solve this problem, we compute the derivative of the curvature instead of the curvature itself.

2. Steering Function

The best method for smooth vehicle navigation known so far by us is to compute the derivative of curvature, dk / ds , as a function of the geometric information and the mission (This function is called *steering function*). After computing this value $dk / ds = f$, the curvature k is updated through the incremental movement Δs . As long as f is a finite value, this method always gives a smooth trajectory in any circumstance. In the mathematical model, we understand the vehicle's curvature is not rapidly changed, hence we include k in the vehicle's configuration. A configuration q is a triple (p, θ, k) of position, orientation, and curvature. We have found the following steering function works in all situations we have applied:

$$\frac{dk}{ds} = -(A\Delta k + B\Delta\theta + C\Delta d) \quad (\text{Eq 3.1})$$

In Eq 3.1, A , B , and C are positive constants which are related to the smoothness of robot's motion [19]. The meanings of these variables, Δk , $\Delta\theta$, and Δd , are as follows: Δk is the difference between the current vehicle's curvature k and the desired curvature k_d . $\Delta\theta$

is the difference between the current vehicle's orientation θ and the desired orientation θ_d . Δd is the vehicle's position error (The exact equation is determined by situations. For instance, if the robot is tracking a directed reference path, it is the "signed" distance from the vehicle position to the directed path).

All geometric information obtained by a robot must eventually be converted into a triple $\Gamma \equiv (\Delta k, \Delta \theta, \Delta d)$ of the curvature difference, orientation difference, and position error. In most geometric situations, this interpretation task is surprisingly straightforward. Once this information Γ is obtained, the robot's motion can be controlled as desired.

3. Line Tracking with Steering Function

a. Introduction

The steering function smooths the line tracking motion. In the line tracking motion, the configuration where the tracking motion starts is called **start configuration**, denoted by $q_s = (p_s, \theta_s, k_s)$. The line that is the goal of line tracking is called the **reference line**, denoted by $q_g = (p_g, \theta_g, k_g)$. We realize that a straight line has zero curvature. In this dissertation we focus on straight line tracking. Thus, the third element in the configuration which specifies a line is normally set to zero. The closest distance [19] from the point p_s , where the tracking motion starts, to the reference line is called the **initial vertical distance**, denoted by d_{init} . The length of the reference line from the point, which is defined by the image of the starting point p_s on the line, to the point, where the tracking trajectory converges to the line, is called **convergence length**, denoted by L . Figure 3.11 illustrates an example of line tracking using steering function. The line tracking can be performed by robot either forward or backward. In forward line tracking the robot advances forward each step in the sense of robot's heading orientation. This is the most common use of line tracking. This section describes the general idea of line tracking. The detailed tracking technique was described as path tracking in [20] [21].

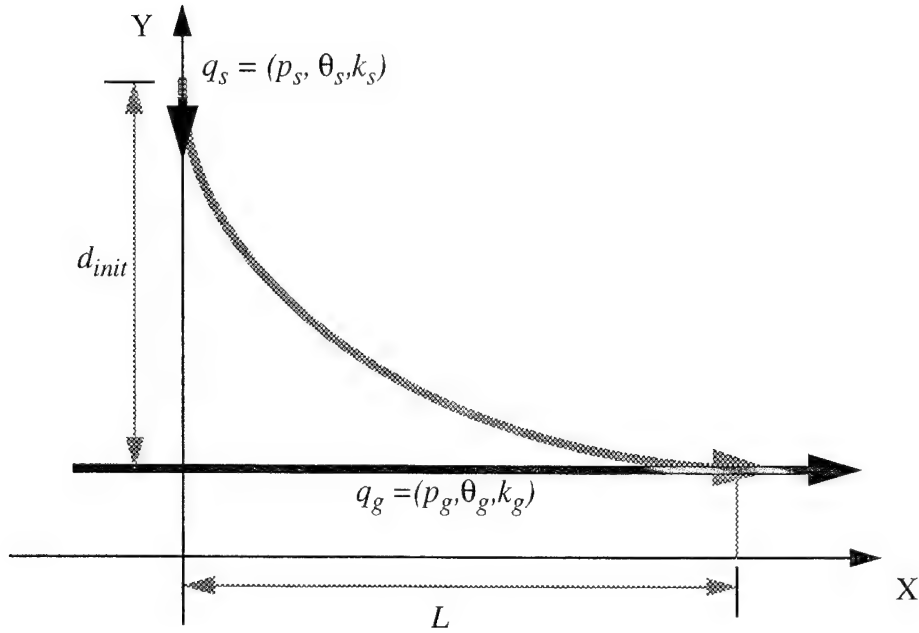


Figure 3.11: The Example of Line Tracking

b. Two Important Types of Line Tracking

Line tracking can be performed between an arbitrary start configuration and a reference line. We are interested in two special pairs of start configuration and reference line. They are considered as two basic types of line tracking. The first type is **parallel line tracking**, in which the orientation of the start configuration is equal to that of the reference line as in Figure 3.12. The second type is **perpendicular line tracking**, in which the orientation of the start configuration is perpendicular to the orientation of the reference line, as Figure 3.13. If the task requires the tracking motion converge to the reference line in a K-region, the convergence length is limited. The limited convergence length is denoted as $L_{allowed}$ in both Figure 3.12 and Figure 3.13. For parallel line tracking, the minimum length of convergence is $L_{allowed} \geq 2.02 d_{init}$ (see Eq 4.9 and the analysis in Chapter IV). For perpendicular line tracking the minimum length of convergence is $L_{allowed} \geq 3.38 d_{init}$ (see Eq 4.10).

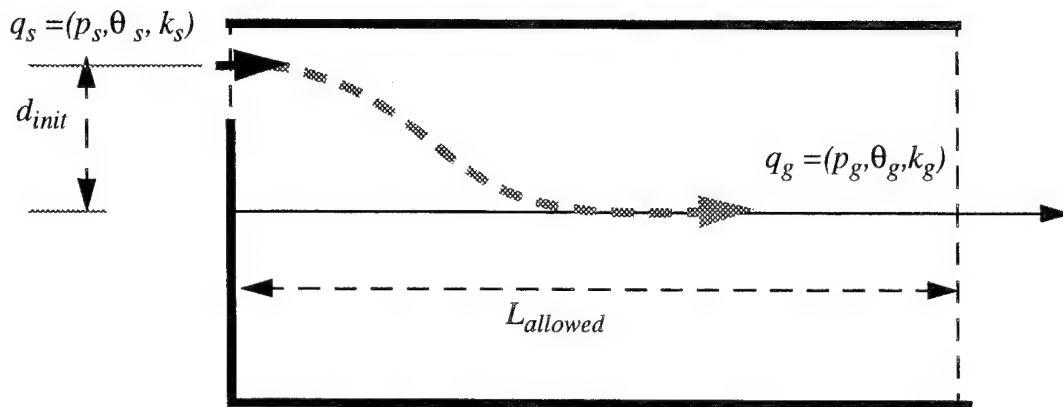


Figure 3.12: The Example of Parallel Line Tracking in K-region.

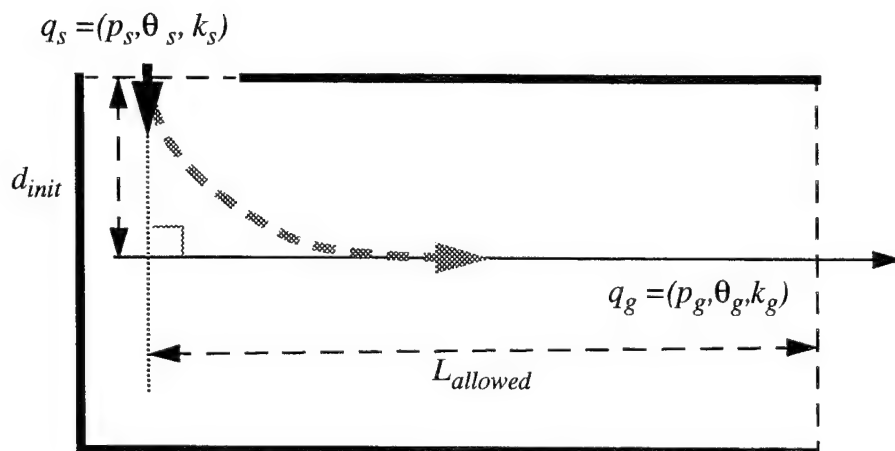


Figure 3.13: The Example of Perpendicular Line Tracking in K-region.

E. FORERUNNER SIMULATION

1. Overview

A **forerunner** is a virtual robot which is used to simulate a real robot's behavior. A simulation that uses the forerunner(s) to simulate the robot's motion in path tracking is called

Forerunner Simulation. Forerunner Simulation can be applied to motion planning and motion control in many aspects. For instance, it can be used to determine the leaving point in the line-to-line transitioning. Using Forerunner Simulation in transition point computation is very meaningful in real-time robot motion control. We will describe this application in Chapter VII. Forerunner Simulation is especially helpful in local motion planning. In this section the mechanism of constructing a Forerunner Simulation will be discussed.

As we know, Forerunner Simulation uses a virtual robot to simulate a real robot's motion. Thus it must perform all computations that are taken in a real-time robot control system, except the portion that is tightly related to the robot's hardware. In this dissertation, the real-time robot control system refers to the Model-based Mobile robot Language (MML) which will be introduced in Chapter VII, VIII, and Chapter IX.

2. Forerunner Simulation Structure

The steering function described in Section A of this chapter and the Appendix A is fundamental component in MML, and is naturally inherited to the Forerunner Simulation. The framework of motion control is the use of a steering function to compute the required curvature change in each motion cycle. The curvature change is in turn used to compute the forerunner's new position. Given an initial configuration $q_{int} = ((x_{int}, y_{int}), \theta_{int}, k_{int})$ as the forerunner's current configuration $q = ((x, y), \theta, k)$ and a reference path of configuration $path = ((x_p, y_p), \theta_p, k_p)$. Based on this frame work, the Forerunner Simulation is constructed by computing the following elements:

- Constants of Steering Function
- Curvature Change
- New Curvature and Orientation Change
- New Configuration of Next Step

a. Constants of the Steering Function

The constants A, B, C of the steering function are determined by the motion smoothness σ as follows:

$$k = 1 / \sigma \quad (\text{Eq 3.2})$$

$$A = 3 k \quad (\text{Eq 3.3})$$

$$B = 3 k^2 \quad (\text{Eq 3.4})$$

$$C = k^3 \quad (\text{Eq 3.5})$$

Because the smoothness is a variable, it can be changed at any time. Especially when dynamic smoothness strategy is applied, the smoothness is determined by the environment, for instance by the size of the K-region. Thus the constants A, B and C have to be computed whenever the smoothness σ is changed.

b. Curvature Change

For the current configuration $q = ((x, y), \theta, k)$, the reference path configuration $path = ((x_p, y_p), \theta_p, k_p)$, and constants A, B, and C, we compute the curvature change as follows:

$$\Delta d = (y - y_p) * \cos(\theta_p) - (x - x_p) * \sin(\theta_p) \quad (\text{Eq 3.6})$$

$$dk / ds = - (A (k - k_p) + B (\theta - \theta_p) + C \Delta d) \quad (\text{Eq 3.7})$$

c. New Curvature and Orientation Change

Once the curvature change dk / ds has been computed, we obtain the new curvature and orientation change by following calculations:

$$k_{new} = k + (dk / ds) * \Delta s \quad (\text{Eq 3.8})$$

$$\Delta \theta = k_{new} * \Delta s \quad (\text{Eq 3.9})$$

here Δs is the transitional length the forerunner advances in each sampling step and this value is given by the forerunner designer. Normally Δs is positive for forward forerunner to increment its next position forward. The value $\Delta \theta$ represents the estimated orientation change between the current position and next position.

d. New Configuration of Next Step

Given the incremental transitional length Δs and the estimated orientation change $\Delta\theta$, we can compute the relative configuration $q_r = ((x_r, y_r), \theta_r, k_r)$ (with respect to the local coordinate system of the current configuration $q = ((x, y), \theta, k)$) of the new configuration in next sampling step where [19]:

$$x_r = (1 - \Delta\theta^2 / 6) * \Delta s \quad (\text{Eq 3.10})$$

$$y_r = (1 - \Delta\theta^2 / 12) * (\Delta\theta / 2) * \Delta s \quad (\text{Eq 3.11})$$

$$\theta_r = \Delta\theta \quad (\text{Eq 3.12})$$

$$k_r = 0.0 \quad (\text{Eq 3.13})$$

The new configuration $q_{next} = ((x_{next}, y_{next}), \theta_{next}, k_{next})$ of the next sampling step is then computed by composing the current configuration q and relative configuration q_r as follows:

$$x_{next} = x + \cos(\theta) * x_r - \sin(\theta) * y_r \quad (\text{Eq 3.14})$$

$$y_{next} = y + \sin(\theta) * x_r + \cos(\theta) * y_r \quad (\text{Eq 3.15})$$

$$\theta_{next} = \theta + \theta_r \quad (\text{Eq 3.16})$$

$$k_{next} = k_{new} \quad (\text{Eq 3.17})$$

3. Algorithms

With these elements, a forerunner simulation template is constructed by the following algorithms as shown in Figure 3.14 and Figure 3.15.

As previously mentioned, the Forerunner Simulation can be applied to many different applications. When the simulation is developed for an application with a specific purpose, the code for performing the application and related preWORK and postWORK will be inserted into the template. The condition for the while loop and desired return value will be specified also.

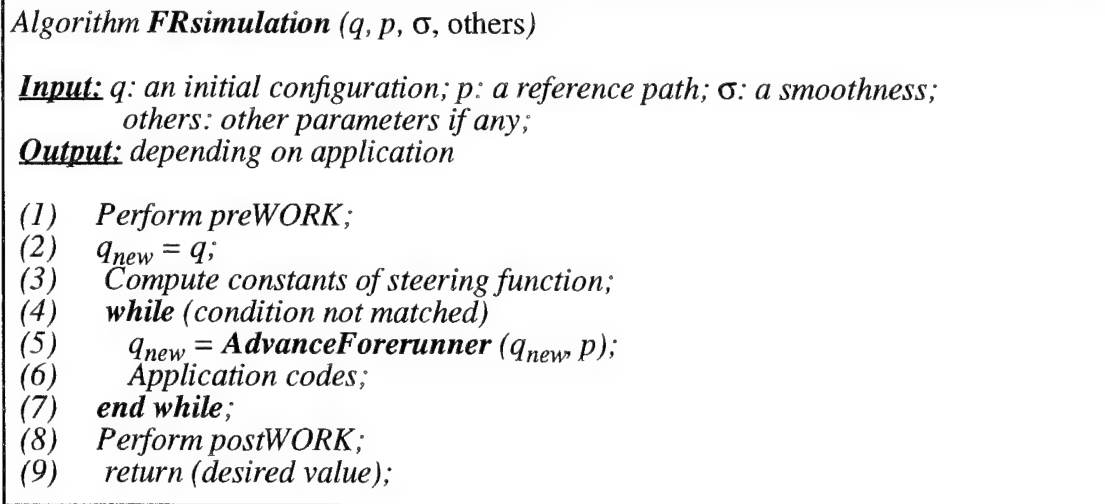


Figure 3.14: The Algorithm for Forerunner Simulation Template.

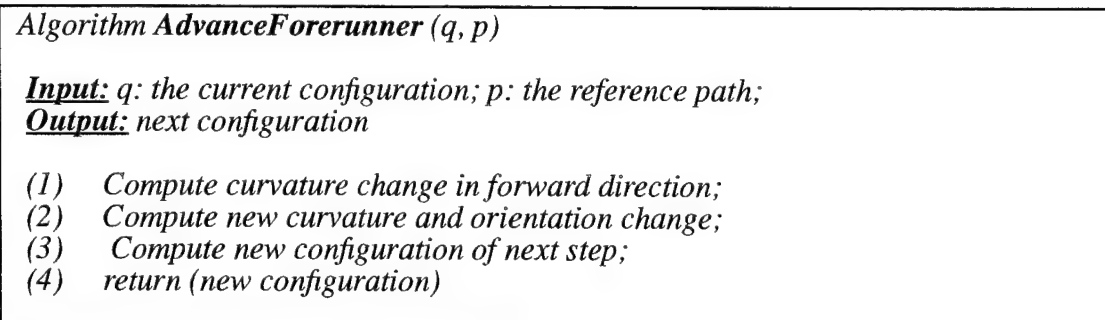


Figure 3.15: The Algorithm for Advancing Forerunner in One Step.

F. SYMMETRIC MOTION PLANNING AND REVERSE PATH

For a configuration $q = (p, \theta, k)$, its **reverse configuration** $rev(q)$ is defined by a tuple as $(p, \pi + \theta, -k)$. Let $q_1 = (p_1, \theta_1, k_1)$ and $q_2 = (p_2, \theta_2, k_2)$ be two distinct configurations on the world. A motion planning method is said to be **symmetric** if, for any configuration q_1 and q_2 , the trajectory of motions planned from q_1 to q_2 is exactly the same as the trajectory

of motions planned from $rev(q_2)$ to $rev(q_1)$. Figure 3.16 illustrates an example of symmetric motion planning corresponding to Figure 3.5.

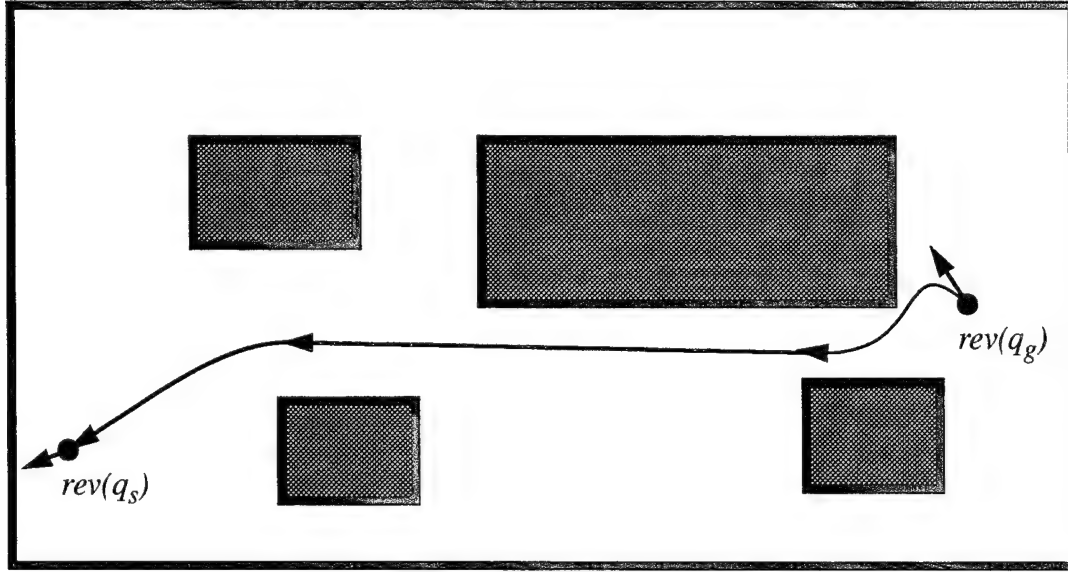


Figure 3.16: A Symmetric Motion Plan

The steering function is a powerful tool for producing a smooth trajectory while tracking a path. For smoothness considerations, when using the steering function to track a path, the trajectory moves toward the reference path quickly at the beginning, but it slows down when the trajectory is getting closer to the path. Thus a symmetric motion planning cannot be done by merely applying path tracking to reverse configurations with the steering function.

For example, in Figure 3.17, the start configuration is $q_s = (p_s, \theta_s, k_s)$. The goal is to track the line specified by $q_g = (p_g, \theta_g, k_g)$, finally stopping at or passing through the configuration q_g . The trajectory converges to the line at the configuration $q_c = (p_c, \theta_c, k_c)$. Now tracking the line specified by configurations $rev(q_s)$ from the configuration $rev(q_g)$, we will have a trajectory as Figure 3.18. The motion plan in Figure 3.17 is obviously not

symmetric to that of Figure 3.18, even though their two end configurations might be the same.

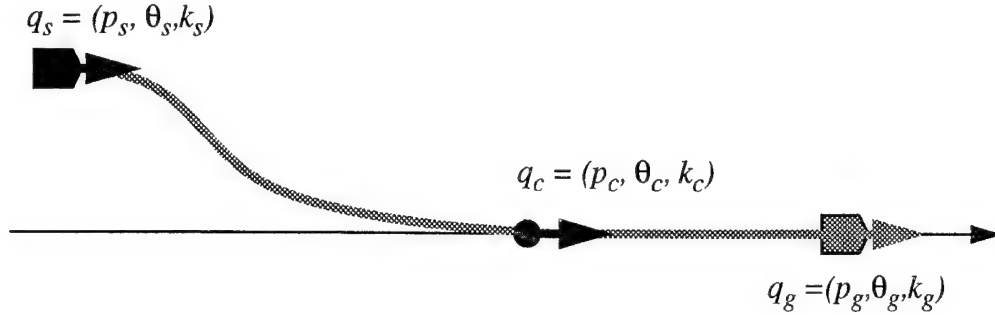


Figure 3.17: A Motion Planned with a Line Tracking.

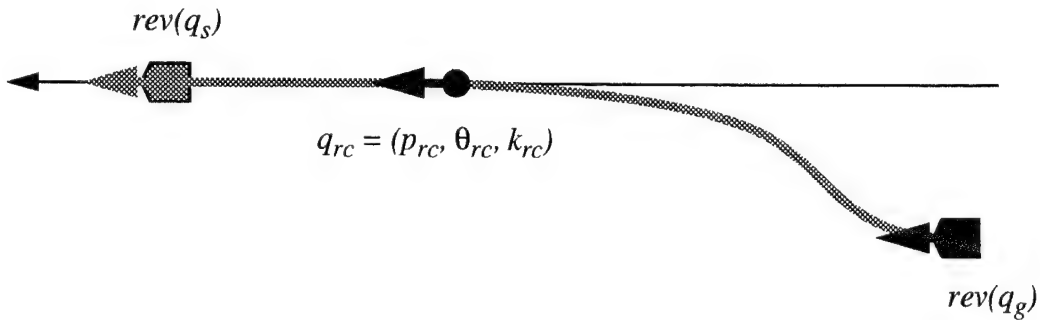


Figure 3.18: A Reverse Line Tracking

The trajectory in Figure 3.17 can be produced in real time by tracking a line specified by the configuration q_g from the configuration q_s . But in real time there is no way to produce such a trajectory by simply applying line tracking from $rev(q_g)$ to $rev(q_s)$. Fortunately, the forerunner simulation can be used to solve the symmetric motion planning problem. As mentioned, the forerunner is a virtual robot which can be used to simulate a real robot's behavior. Again we assume q_s and q_g are the start and goal configurations respectively. For symmetric motion planning, we set the forerunner at $rev(q_g)$ to perform path tracking with $rev(q_s)$ as its reference line. The simulated trajectory can be stored as a

sequence of configurations. A **reverse path** can be constructed by reversing each configuration and rearranging configurations in the sequence with opposite order. Then the symmetric motion planning involves generating and tracking reverse paths. The reverse path is formulated in the following two steps:

- Step 1. Set the forerunner to the reverse configuration of goal configuration q_g and run forerunner simulation to track a line specified by reverse configuration of start configuration q_s . While the forerunner is traveling, store the trajectory (by configurations) until the tracking converges to the reference line. We represent this path with a sequence of configurations as:

$$C = \langle q_1, \dots, q_i, \dots, q_n \rangle \quad \text{where } n \geq 1 \text{ and } q_i = (p_i, \theta_i, k_i) \text{ for all } 1 \leq i \leq n;$$

- Step 2: Reverse the entire sequence to generate a reverse path as follows:

$$C_{rev} = \langle q_n, \dots, q_i, \dots, q_1 \rangle \quad (\text{Eq 3.18})$$

where $n \geq 1$ and $q_i = (p_i, \pi + \theta_i, -k_i)$ for all $1 \leq i \leq n$;

Ideally, the first configuration q_n in path C_{rev} should specify a line that is identical to the line specified by the start configuration $q_s = (p_s, \theta_s, k_s)$.

If a local motion planning method is symmetric, reverse paths will be generated for many cases. Circumstances under which a reverse path needs to be generated will be discussed in Chapter IV and V.

G. REVERSE PATH TRACKING

The normal path tracking provides a technique for vehicle odometry correction to catch up to the reference path. The reference path is normally specified by a configuration, and the vehicle odometry error can be computed based on that configuration. A reverse path consists of a sequence of configurations, as shown in Eq 3.18. Each configuration in the sequence can be a reference path element to the vehicle when the vehicle is tracking the reverse path. The difference between two consecutive configurations is normally small in all elements of the configuration. While the vehicle is moving, determining which configuration in the sequence is to be taken as a reference path becomes critical issue. In

order to track the reverse path as precisely as possible, it is necessary to have a new path tracking technique other than the normal one. This section addresses how the reverse path tracking can be performed properly. We begin by determining the path tracking smoothness.

Since the difference between consecutive configurations is small, the vehicle's turning motion needs to be sharpened so that it can converge to the reference path faster. Therefore, the smoothness, σ , of the steering function in reverse path tracking should be smaller than the smoothness, σ_g , used to generate that reverse path. Our experiments showed that the proper smoothness for reverse path tracking is as Eq 3.19:

$$\sigma = \sigma_g / 2 \quad (\text{Eq 3.19})$$

Let's assume that a reverse path is generated by forerunner as illustrated in Figure 3.19. In the figure, the reverse path Π consists of the configurations q_1, \dots, q_6 in sequence (this is for purpose of illustration; the actual sequence will be much denser to specify the path more precisely), and $q_v = (p_v, \theta_v, k_v)$ is the vehicle's current configuration. A global coordinate system (GCS) is defined as shown in the figure. To track this path Π smoothly by using the steering function, the vehicle takes the first configuration $q_1 = (p_1, \theta_1, k_1)$ as the initial reference path. Then the differences of $\Delta\theta$, Δk , and Δd are calculated as follows:

$$\Delta\theta = \Delta\theta_s - \Delta\theta_1 \quad (\text{Eq 3.20})$$

$$\Delta k = \Delta k_s - \Delta k_1 \quad (\text{Eq 3.21})$$

$$\Delta d = y^* \quad (\text{Eq 3.22})$$

The signed distance value y^* in Eq 3.22 is the shortest distance between the vehicle's current configuration and reference path [20]. These differences, $\Delta\theta$, Δk and Δd , are applied to the steering function to determine vehicle's necessary curvature in the next control cycle so that the vehicle is made to travel along the reference path smoothly.

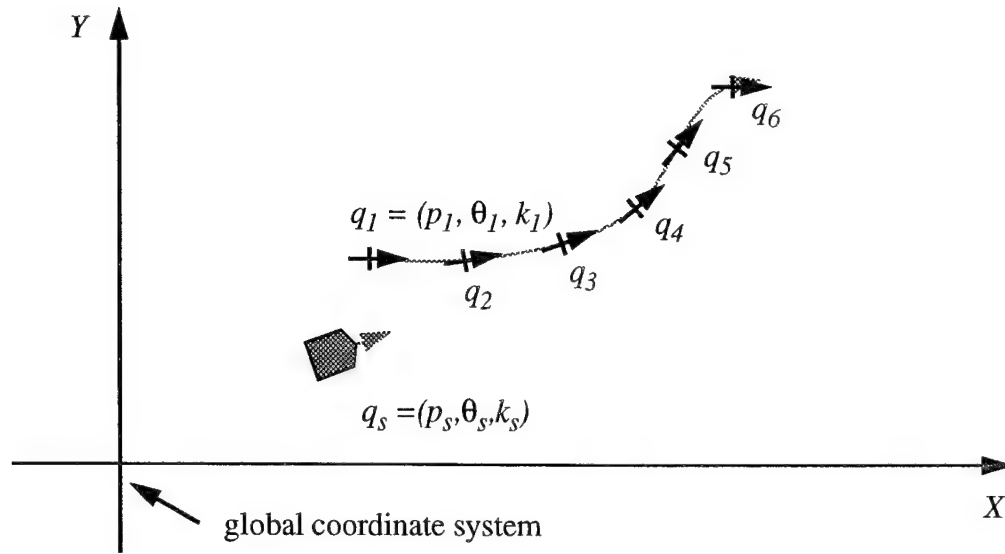


Figure 3.19: A Reverse Path

As the vehicle is moving along the path specified by a give configuration, a critical decision must be made to allow the vehicle to transition from the current reference path to the next. There is a simple way to determine when to transition, as described below. This method uses the relative position of the vehicle's current configuration and the configuration that specifies the current reference path element to determine whether the vehicle has passed the position of reference path. For this purpose, a local coordinate system (LCS) is defined as illustrated in Figure 3.20. The LCS takes the center of the vehicle as its origin. A line, which passes the origin and is parallel to vehicle's global orientation, defines the positive x axis of the LCS. The positive y axis of LCS is then defined by a perpendicular line (to the x axis), passing through the local origin and heading to the vehicle's left as shown in Figure 3.20.

When the local coordinate system is established, the configuration of reference path can be transformed from global GCS to LCS. The computation of the configuration transformation is as following. Let $q_v = (p_v, \theta_v, k_v)$ and $q_{ref} = (p_{ref}, \theta_{ref}, k_{ref})$ be the vehicle's

current configuration and reference path configuration in GCS respectively. Assume that the reference path configuration q_{ref} with respect to LCS is $q^* = (p^*, \theta^*, k^*)$. Then we have

$$q_v \circ q^* = q_{ref} \quad (\text{Eq 3.23})$$

$$q_v^{-1} \circ q_v \circ q^* = q_v^{-1} \circ q_{ref} \quad (\text{Eq 3.24})$$

$$q^* = q_v^{-1} \circ q_{ref} \quad (\text{Eq 3.25})$$

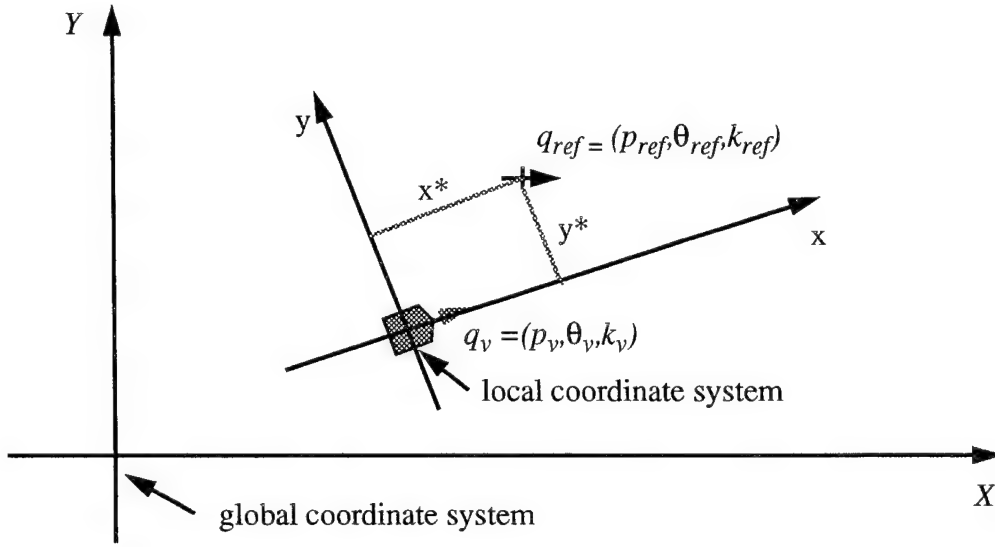


Figure 3.20: Local Coordinate System on Reverse Path Tracking

In computing q^* , we will simplify the configurations to transformations [20]. Thus, the transformation $q^* = (x^*, y^*, \theta^*)^T$ stands for configuration $q^* = (p^*, \theta^*, k^*)$. Transformation $q_v = (x_v, y_v, \theta_v)^T$ stands for configuration $q_v = (p_v, \theta_v, k_v)$ and transformation $q_{ref} = (x_{ref}, y_{ref}, \theta_{ref})^T$ stands for configuration $q_{ref} = (p_{ref}, \theta_{ref}, k_{ref})$. Then we have

$$\begin{aligned}
q^* &= q_v^{-1} o q_{ref} = \begin{bmatrix} x_v \\ y_v \\ \theta_v \end{bmatrix}^{-1} o \begin{bmatrix} x_{ref} \\ y_{ref} \\ \theta_{ref} \end{bmatrix} = \begin{bmatrix} -x_v \cos \theta_v - y_v \sin \theta_v \\ x_v \sin \theta_v - y_v \cos \theta_v \\ -\theta_v \end{bmatrix} o \begin{bmatrix} x_{ref} \\ y_{ref} \\ \theta_{ref} \end{bmatrix} \\
&= \begin{bmatrix} (x_{ref} - x_v) \cos \theta_v + (y_{ref} - y_v) \sin \theta_v \\ -(x_{ref} - x_v) \sin \theta_v + (y_{ref} - y_v) \cos \theta_v \\ \theta_v - \theta_{ref} \end{bmatrix}
\end{aligned} \tag{Eq 3.26}$$

The signed valued $x^* = (x_{ref} - x_v) \cos \theta_v + (y_{ref} - y_v) \sin \theta_v$ can be used to determine when to transition to the path specified by the next configuration in the sequence. When x^* becomes negative, it means that the vehicle has passed the position of the current reference path segment. Then the next configuration is taken as reference path.

H. LOCAL MOTION PLANNING STEPS

The local motion planning is executed in the following steps:

- Compute intermediate configurations
- Perform end-portion motion planning.
- Perform mid-portion motion planning.

The first step can be viewed as a preprocessing of local motion planning. The purpose of preprocessing is to make local motion planning standardized and simplified. In this section we discuss the preprocessing step, and an overall algorithm for local motion planning is provided. We will present the details of end-portion and mid-portion motion planning to Chapter IV and V.

The local motion planner performs local motion planning region-by-region. The means of connecting the motion in consecutive regions becomes more important at this moment. The exit border of a region is the entrance border of next region in the global path class. Thus, a border can serve as the interface of two regions' motion communication. We define an intermediate configuration for each border in the crossing sequence using the border's crossing point and orientation so that the intermediate configurations in a region

can be used to plan robot's motion in that region while connecting the motion to the next region. How to determine the crossing points on borders will be discussed in this section. In order to standardize and simplify the local motion planning, an orthogonalized border orientation needs to be computed for those borders that are not parallel to X or Y axis. After the crossing point and orientation of a border are determined, an intermediate configuration for that border can be defined. For example, if p represents the crossing point obtained and θ represents the border orientation, then the configuration q is defined as $q = (p, \theta, k)$, where curvature $k = 0$.

1. Determination of Crossing Position on the Border

Since the robot's motions in neighboring regions of the path are related, it is necessary to consider the motion in the region that follows the current one. The crossing point on a border actually connects and controls the motion on the two regions that shares the common border. Thus, determining where to cross the border between two regions is important to the local motion planning. A proper position to cross the border can make local motion planning easier in each region. In this dissertation, safety is the most important factor in local motion planning. Therefore, the crossing point on the border must be in a range that allows the vehicle to cross the border without collision. A border's safe crossing range depends upon the vehicle's width and the minimum clearance (if required). The global path class for motion planning is represented by a sequence of all passable regions accompanied by their related borders. This implies that every border has a safe range. Therefore, crossing a border at its center (mid-point) will be the most desirable plan in the sense of safety. We will apply this idea in the local motion planning in the following chapters. However, there is an exception in which we do not take the center of the border as a crossing position. That is in some special regions with parallel entrance and exit borders and in which following situations exist:

- At least one of the centers of the entrance and exit borders has its image (a perpendicularly projected point) on another border in the safe range.

- $FL < 2.02 * d$, where FL is the forward length of the region and d is the distance between centers of the two borders measured in the direction perpendicular to the entrance orientation.

In such a region, keeping the strategy of crossing the border at its center will make local motion planning complicated and is undesirable, because for these two parallel configurations there is not enough length for parallel line tracking from the entrance configuration to converge to the exit configuration (see Eq 4.9 in Chapter IV Section B).

Figure 3.21 illustrates a possible region in a global path class. In the figure, the region, R_k , is the region that needs to be considered specially. The solution, which is to relocate the crossing point of the border at the image of another border's center, is also shown in the figure.

The reasons for relocating the crossing point in this kind of region are because entrance and exit borders are parallel, and the region has not enough length in the forward direction to perform a complete parallel line tracking (from entrance configuration to exit configuration and converge in the region). Under this situation the local motion planning will be forced to plan the motion using other than parallel-line tracking. Thus, relocating the crossing point on E-border makes two crossing points in a straight line which is exactly on the exit configuration. The motion in this region then will be simpler and more desirable.

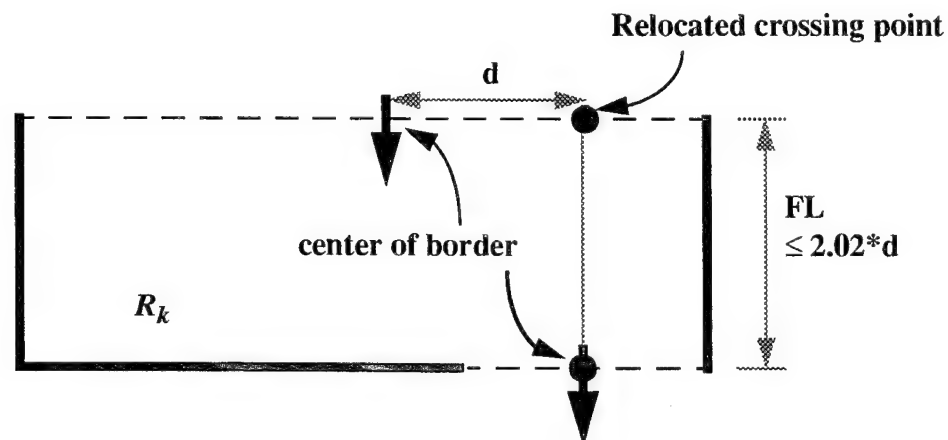


Figure 3.21: Relocating Border Crossing Point in an Exceptional Region.

2. Orthogonalizing Border Orientations

As we realize, K-region decomposition allows non-orthogonal borders in regions as illustrated in Figure 3.1. In Figure 3.1, the region R_6 and R_3 share a non-orthogonal border B_6 . The orientation of border B_6 can be orthogonalized to its nearest orthogonal orientation as illustrated in Figure 3.22. Orthogonalizing the orientations of non-orthogonal borders makes the region standardized in the sense of orthogonal entrance and exit borders. The local motion planning then can be performed by simply using parallel line tracking and perpendicular line tracking techniques.

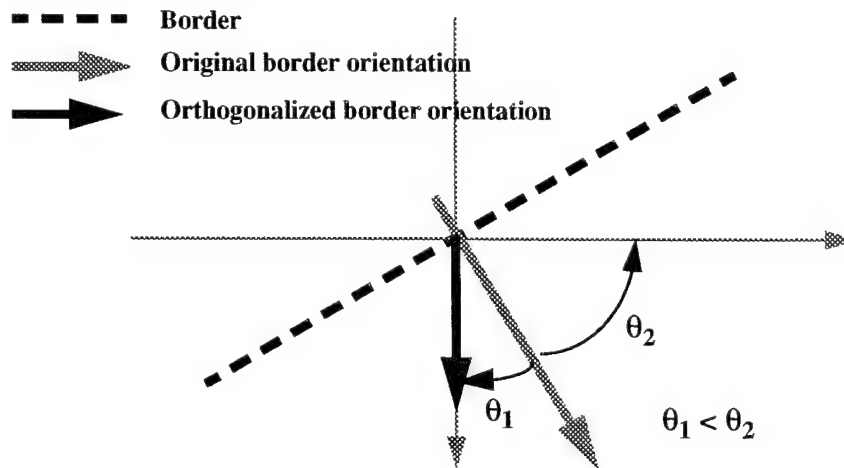


Figure 3.22: Orthogonalizing the Orientation of a Border

3. Algorithm

Naturally, the local motion planning is conducted in the following order:

- Compute intermediate configurations
- Perform initial motion planning.
- Perform mid-portion motion planning.
- Perform final motion planning.

This is under the assumption that all end-portion motion planning can be done in initial and final regions. Under this assumption, the mid-portion motion planning is carried out region-by-region from the second region of the global path class to the region next to the final region on the path. The following algorithm for local motion planning is based on this assumption. If the final motion planning involves more than one region, the order of planning needs to be modified to let the end-motion planning be finished before mid-portion motion planning starts.

Algorithm LocalMP (q_s, q_g, Π, W)

Input: q_s : start configuration;
 q_g : goal configuration;
 Π : a crossing sequence;
 W : a world model

Output: Motion planning data structure

```

(1)  if (length ( $\Pi$ ) = 1) then
(2)    SingleRegionMP ( $q_s, q_g, W$ );
(3)  else
(4)     $Q = Q_0 = \text{ComputeIntConfig}(\Pi, W)$ ;
(5)     $q = q_s$ ;
(6)     $MP = \text{null}$ ;
(7)     $q_{\text{exit}} = \text{car}(Q)$ ;
(8)     $r_{\text{current}} = \text{car}(\Pi)$ ;
(9)    append ( $MP, \text{InitialMP}(q, q_{\text{exit}}, r_{\text{current}}, W)$ );
(10)    $q = q_{\text{exit}}$ ;
(11)    $\Pi = \text{cdr}(\text{cdr}(\Pi))$ ;
(12)    $Q = \text{cdr}(Q)$ ;
(13)    $r_{\text{current}} = \text{car}(\Pi)$ ;
(14)   while (not empty( $Q$ ))
(15)      $q_{\text{exit}} = \text{car}(Q)$ ;
(16)     append ( $MP, \text{MidPortionMP}(q, q_{\text{exit}}, r_{\text{current}}, W)$ );
(17)      $q = q_{\text{exit}}$ ;
(18)      $\Pi = \text{cdr}(\text{cdr}(\Pi))$ ;
(19)      $Q = \text{cdr}(Q)$ ;
(20)      $r_{\text{current}} = \text{car}(\Pi)$ ;
(21)   end while ;
(22)   append ( $MP, \text{FinalMP}(q, q_g, r_{\text{current}}, W)$ );
(23) end if;
```

The subroutine **SingleRegionMP** is a motion planning subroutine for a global path class having only one region, i.e. the start and goal configurations reside in the same region. Subroutine **ComputeIntConfig** includes two steps. First, it determines the crossing point of each border by the rule described earlier in this section. Second, it orthogonalizes the orientations of borders. This part is also described in this section. Finally, the subroutine defines configurations for each border and stores the configurations in *Q*. The algorithm for this subroutine is not presented in this dissertation. Subroutine **InitialMP** and **FinalMP** are for end-portion motion planning which will be described in Chapter V. The subroutine **MidPortionMP** is a mid-portion motion planning subroutine which is presented in Chapter IV. There some other utility functions that are not presented in the dissertation because their functionality is pretty straightforward. The meanings of the functions are explained as follows. The function *length* returns the number of elements (including regions and borders) left in the global path class. The function *append* appends a planned motion instruction to a data structure *MP*. The function *car* returns the first element of the object passed in, while *cdr* returns all element but the first one.

IV. MID-PORITION MOTION PLANNING

A. INTRODUCTION

The fact that the K-regions are convex polygons makes the mid-portion motion planning simple and straightforward. Nevertheless, the motion planning problem has complex aspects. Safety is one of the most important concerns in motion planning at this stage. The safest path in a region is the path that follows the Voronoi boundary [22], because this path stays equidistant among the closest objects. Unfortunately, most Voronoi boundaries cannot be a part of a feasible path for autonomous vehicles with kinematic and nonholonomic constraints. However, it gives us the idea that the motion will be considered safer if it stays further away from objects. Therefore, we propose a method in this chapter which plans the robot's motion along the global path class with its trajectory as close to the center line of the regions as possible. How to plan such motions for a rigid-body mobile robot is the main subject to be discussed in this chapter.

1. Problem Statement

The problem to be solved in mid-portion motion planning is as follows:

Given a world model, W , on a two-dimensional plane, \mathbb{R}^2 , and a global path class $\Pi = \langle R_{i1}, B_{i1}, \dots, R_{i(n-1)}, B_{i(n-1)}, R_{in} \rangle$, with R_{is} , R_{it} indicating the first region and the last region which are considered mid-portion of the global path class. The mid-portion motion planning is to plan a safe motion symmetrically for a rigid body robot to travel from the entrance configuration of region R_{is} to the exit configuration of region R_{it} along the global path class Π .

The inputs to the mid-portion motion planner are the world model W , the global path class Π and the regions R_{is} , R_{it} which indicate the first region and the last region of mid-portion of the global path class. The outputs are a sequence of motions (Figure 4.1).

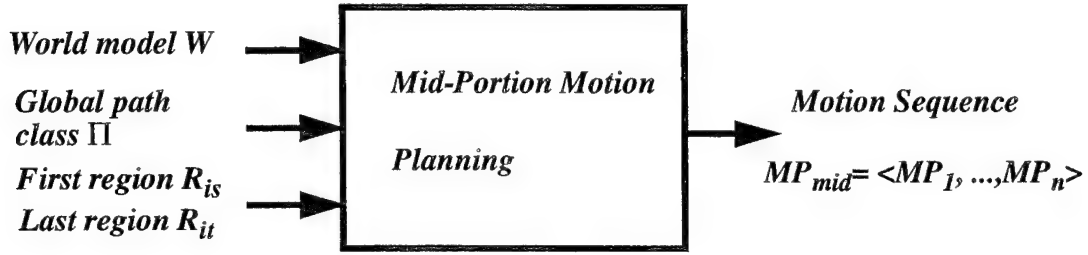


Figure 4.1: The Mid-Portion Motion Planning

2. Definitions

The crossing points of entrance and exit borders, P_{in} and P_{out} , and the border orientations Ψ_{in} and Ψ_{out} define the entrance configuration, $q_{in} = (P_{in}, \Psi_{in}, 0)$, and exit configuration, $q_{out} = (P_{out}, \Psi_{out}, 0)$, which actually specify the lines crossing the entrance border and exit border with border orientations. Although most of the crossing points are at the centers of borders, there are a few exceptions which determine other points on the borders as the crossing points in the preprocessing of local motion planning. The definition of entrance and exit configurations will be intensively used in this chapter and in the chapters that follow.

In order to describe the local motion planning clearly, we define some additional symbols to represent different distance measurements in a K-region based on the entrance and exit borders as Figure 4.2. In this figure, the line that is the bisector of two Forward Edges is called **Horizontal Center Line**, denoted by **HCLine** or **HCL**. The line that is the bisector of two Cross Edges is called **Vertical Center Line**, denoted by **VCLine** or **VCL**. We can use configurations to specify the enter lines. For instance, the Horizontal Center Line can be specified by the configuration $q_{HCLine} = (EC_{in}, \Psi_{in}, 0)$, where EC_{in} is the center of Entrance Edge and Ψ_{in} is the orientation of entrance border. The center line specified by

q_{HCLine} can be the same as that specified by q_{in} , depending on whether the centers of borders coincide with centers of its corresponding edges.

By d_H we denote the distance between two points projected by the crossing points of the exit border and entrance border on the Vertical Center Line. By d_{Hin} and d_{Hout} we denote the closet distance from the crossing points of the entrance and exit borders, P_{in} and P_{out} to the Horizontal Center Line, respectively. Similarly, d_V denotes the distance between two points projected by the crossing points of exit border and entrance border on the Horizontal Center Line. By d_{Vin} and d_{Vout} we denote the closest distance from the crossing point of entrance and exit borders P_{in} and P_{out} respectively, to the Vertical Center Line. The value of the d_H may be equal to the sum of d_{Hin} and d_{Hout} , but not always. For instance, in Figure 4.2, they are equal. If the entrance and exit borders are on the same side of Horizontal Center Line, they will be not equal. The same relationship is found among d_{Vin} , d_{Vout} and d_V .

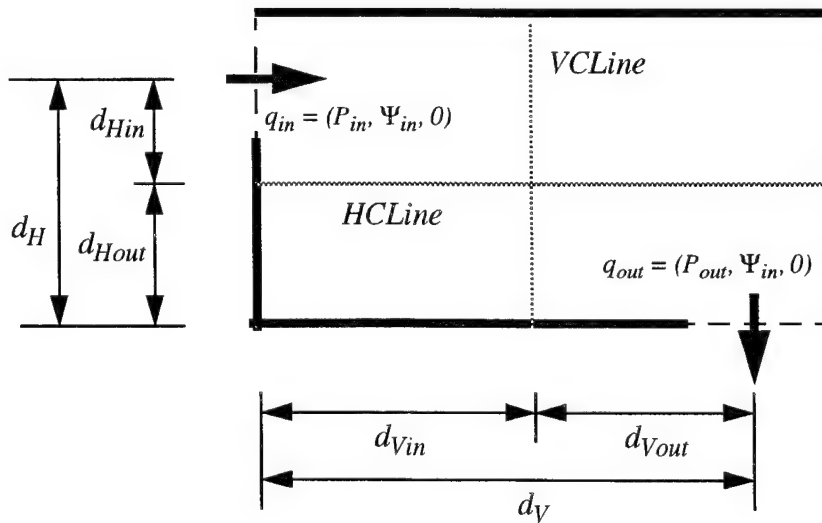


Figure 4.2: The Definition of Distance Measurements in a K-region

B. STEERING FUNCTION CHARACTERISTICS

1. Simulation Results Analysis

Because most of motion in a K-region will be of the form of tracking a reference line that is parallel or perpendicular to the initial configuration, analysis will focus on the simulation of parallel line tracking and perpendicular line tracking. All of the simulations are done under the assumption that the tracking trajectory is considered to have converged if

$$\Delta d \leq d_{init}/1000, \quad (\text{Eq 4.1})$$

where Δd is the closest distance from the current configuration on the tracking trajectory to the reference line and d_{init} is the initial vertical distance.

From simulation data, (Table 4.1, Table 4.2, Table 4.3 and Table 4.4), we found that in line tracking there is a close relationship among the smoothness σ , the vertical distance d_{init} , and the convergence length L . Based on the relationship we found, some rule, that help in computing dynamic smoothness in line tracking can be developed.

Table 4.1: The Relationship among Distances and Smoothness in Parallel Line Tracking (initial $\Delta\theta = 0$) with Minimum Smoothness σ

<i>Initial $\Delta\theta$</i>	<i>d_{init}</i>	<i>σ</i>	<i>L</i>	<i>σ/d_{init}</i>
0	400.0	70.7	695.8	0.18
0	300.0	53.0	521.2	0.18
0	200.0	35.4	347.9	0.18
0	100.0	17.7	173.28	0.18
0	80.0	14.2	138.9	0.18
0	60.0	10.6	103.2	0.18
0	40.0	7.1	68.8	0.18
0	20.0	3.5	33.1	0.18

Table 4.2: The Relationship among Distances and Smoothness in Parallel Line Tracking (initial $\Delta\theta = 0$) with Various Smoothness σ

<i>Initial $\Delta\theta$</i>	<i>d_{init}</i>	σ	L	L / σ
0	20.0	3.6	34.5	9.58
0	20.0	10.0	109.6	10.96
0	20.0	20.0	222.3	11.12
0	20.0	40.0	447.1	11.18
0	20.0	60.0	671.8	11.20
0	20.0	80.0	896.3	11.20
0	20.0	100.0	1120.8	11.21
0	20.0	200.0	2243.8	11.22
0	20.0	400.0	4489.7	11.22

Table 4.3: The Relationship among Distances and Smoothness in Perpendicular Line Tracking (initial $\Delta\theta = \pi / 2$) with Various Smoothness

<i>Initial $\Delta\theta$</i>	<i>d_{init}</i>	σ	L	σ / d_{init}
$\pi / 2$	100.0	22.0	200.0	0.22
$\pi / 2$	100.0	40.0	331.8	0.40
$\pi / 2$	100.0	41.0	334.3	0.41
$\pi / 2$	100.0	42.0	335.5	0.42
$\pi / 2$	100.0	43.0	335.4	0.43
$\pi / 2$	100.0	44.0	333.4	0.44
$\pi / 2$	100.0	45.0	329.1	0.45
$\pi / 2$	100.0	50.0	246.3	0.50

Table 4.4: The Relationship among Distances and Smoothness in Perpendicular Line Tracking (initial $\Delta\theta = \pi / 2$) with Various d_v and Corresponding Maximum Smoothness σ

<i>Initial $\Delta\theta$</i>	<i>d_{init}</i>	σ	L	L / d_{init}
$\pi / 2$	400.0	168.0	1351.7	3.38
$\pi / 2$	300.0	126.0	1012.9	3.38
$\pi / 2$	200.0	84.0	674.3	3.37
$\pi / 2$	100.0	42.0	335.5	3.35
$\pi / 2$	80.0	33.6	267.9	3.35
$\pi / 2$	60.0	25.2	200.0	3.33
$\pi / 2$	40.0	16.8	132.4	3.31
$\pi / 2$	20.0	8.4	64.5	3.23

Several facts are revealed after analyzing those simulation data. First of all, we found that the smoothness applied to the steering function in line tracking can be dynamically determined depending on the following factors:

1. The vertical distance d_{init} .
2. The allowed convergence length L .
3. The initial orientation difference $\Delta\theta = \theta_s - \theta_g$.

From the experiences of using steering function for motion planning over years, we realize that there is no maximum limitation in smoothness of parallel line tracking. However, if the proportion of smoothness to vertical distance is too small, the tracking trajectory will be unreasonable. And in even the worst case, the steering function cannot make the tracking trajectory converge to the reference line. (see Appendix A) The simulation results from Table 4.1 show that for acceptably smooth line tracking, the minimum proportion of smoothness versus various vertical distance is 0.18 as Eq 4.2.

$$\sigma/d_{init} = 0.18 \quad (\text{Eq 4.2})$$

Being aware of the limitation of the minimum proportion, we would like to compute the desirable smoothness that allows the line tracking trajectory to converge in limited length of reference line. Table 4.2 shows a set of simulation results using various smoothness values (greater than or equal to the minimum one). We found that for all possible smoothness values σ with a fixed d_{init} , the converge distances L always satisfies Eq 4.3.

$$L/\sigma \leq 11.22 \quad (\text{Eq 4.3})$$

Eq 4.3 suggests that the maximum proportion of convergence length to smoothness σ is approximately 11.2 for all possible smoothness in parallel line tracking. When the smoothness can be dynamically determined, we want to obtain the smoothness by simple computation. Eq 4.2 and 4.3 are the key factors in determining the smoothness of parallel-line tracking in a specific K-region. We will discuss this in the next subsection.

For perpendicular line tracking, our task is to find the maximum desirable smoothness. The simulation results is summarized in Table 4.3. The table shows that when the vertical distance d_{init} is known, the most desirable (maximum) smoothness which does not lead to an oscillation convergence is as Eq 4.4:

$$\sigma = 0.42d_v \quad (\text{Eq 4.4})$$

Although the Table 4.3 (based on $d_{init} = 100$ cm) is only a part of the simulation results, Eq 4.3 is concluded from all simulations with various d_{init} .

As the basic concept of smoothness shows, in line tracking using steering function, the larger the smoothness is, the longer it takes to converge. When the length of the reference line is limited, the smoothness determined by Eq 4.3 may not be able to make the line tracking converge within the limited range. Thus the constraint of Eq 4.3 should be considered. The Table 4.4 shows the relationship between L and d_{init} when the most

desirable smoothness for each individual d_{init} is applied. From Table 4.4, we observed that the maximum convergence length is given by

$$L = 3.38d_{init} \quad (\text{Eq 4.5})$$

Eq 4.5 can be viewed as a lower bound of convergence length when the vertical distance d_{init} is fixed in perpendicular line tracking. Equations 4.4 and 4.5 are the main factors of determining dynamic smoothness in perpendicular line tracking. We will discuss this in more detail in the next subsection.

2. Dynamic Smoothness

For a specific K-region, motion planning based on line tracking needs to determine a reasonable smoothness so that the motion in that region is as smooth as possible under the safety consideration. We discuss the dynamic smoothness determination in two common types of line tracking in a K-region.

a. Parallel Line Tracking

For the line tracking in a K-region, the length of the reference line is fixed. Thus there is a limitation on converge distance. As previously mentioned, the convergence length L in a region is actually the allowed convergence length $L_{allowed}$ when discussing this subject. We intend to compute the smoothness that is most desirable under this limitation. Since the maximum proportion of convergence length to the smoothness is 11.22 as the conclusion we found in Table 4.1, the smoothness σ can mainly be determined by the allowed convergence length as following:

$$\sigma = L_{allowed}/11.22 \quad (\text{Eq 4.6})$$

However, because of the characteristics of the steering function, the smoothness cannot be too small compared with the initial vertical distance d_{init} in the parallel line tracking. Otherwise its trajectory will be unreasonable. Thus we know the smoothness obtained from Eq 4.2 is the lower bound of all possible smoothness. We then can have the smoothness bounded as Eq 4.7.

$$\sigma \geq 0.18d_{init} \quad (\text{Eq 4.7})$$

By replacing σ of Eq 4.7 with 4.6, we have

$$L_{allowed}/11.22 \geq 0.18d_{init} \quad (\text{Eq 4.8})$$

$$L_{allowed} \geq 2.02d_{init} \quad (\text{Eq 4.9})$$

Therefore the smoothness of parallel line tracking can be computed as Eq 4.6 under the condition of $L_{allowed} \geq 2.02d_{init}$.

The parallel line tracking will mainly be applied in the motion planning in the region with two parallel borders. In most cases, $L_{allowed}$ will be the length of the region FL in forward direction of two parallel borders. This implies that with minimum reasonable smoothness the minimum distance that allows a parallel line tracking to converge is as Eq 4.9. Therefore if $FL = L_{allowed} < 2.02 d_{init}$, it is not possible to track a line that is parallel to the orientation of the entrance/exit border, from the center of entrance border such that the trajectory converges to the line before the point p_{out} of configuration q_{out} . To solve the motion planning problem under this situation, an extra step is needed in such a region. That is if $FL < 2.02 * d_{init}$, we will compute a center line between entrance border and exit border that has its orientation perpendicular to the borders' orientation. The center line is actually the bisector of the entrance and exit borders. Then the motion in that region will become a perpendicular line tracking as Figure 4.3. The computation of perpendicular line tracking will be described in next subsection.

b. Perpendicular Line Tracking

We are always seeking smooth motion in the motion planning. As the analysis in previous subsection, in perpendicular line tracking, the maximum smoothness that does not make perpendicular line tracking motion oscillate is $\sigma = 0.42 * d_{int}$ for a known vertical distance d_{init} . However, as in parallel line tracking, there is a limitation on the length of the reference line that allows perpendicular line tracking motion to converge in a K-region. In order to make perpendicular line tracking possible under this limitation,

the configuration where the line tracking starts needs to be determined prior to the determination of smoothness. The Eq 4.5 gives us the minimum convergence length in general. It can be expressed as following:

$$L_{allowed} \geq 3.38d_{init} \quad (\text{Eq 4.10})$$

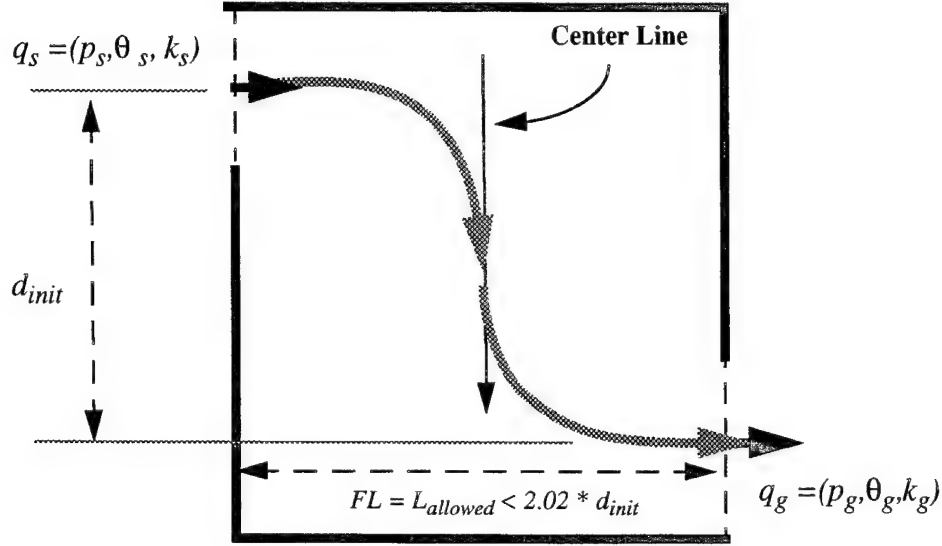


Figure 4.3: An Example of Perpendicular Line Tracking on Center Line in a K-region.

Eq 4.10 must be satisfied in any case if perpendicular line tracking is to be performed. Because the reference line and its length are fixed in a region, the start configuration must be movable to adjust its distance to reference line. We take the configuration q defined by the center of border as the initial position to measure the distance d_{init} . The configuration q can be either q_{in} or q_{out} . Then the start configuration can be decided by following algorithm:

$$q_s = q;$$

Compute d_{init} ;

$$\text{if } (L_{allowed} < 3.38 * d_{init})$$

Compute new start configuration q_s along the line specified by q
such that $d_{init} = L_{allowed} / 3.38$;

Once the start configuration is determined, the smoothness σ can be computed by:

$$\sigma = 0.42d_{init}$$

The Figure 4.4 illustrates the perpendicular line tracking in the case of $L_{allowed} < 3.38 * d_{init}$ at the beginning. The start configuration $q_s = (p_s, \theta_s, 0)$ is moved forward to the new start configuration $q_{new} = (p_{new}, \theta_s, 0)$ to shorten d_{init} so that the condition $L_{allowed} \geq 3.38 * d_{init}$ is satisfied.

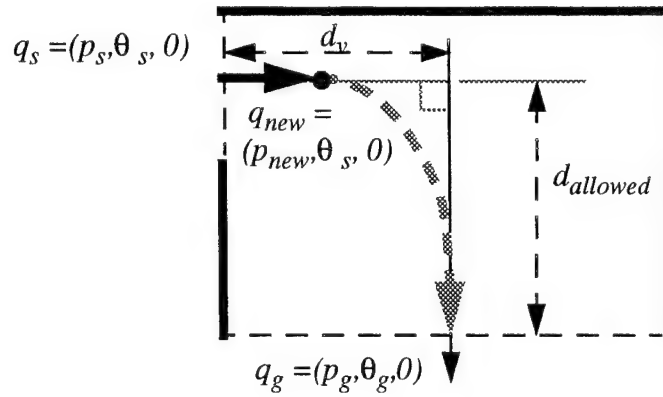


Figure 4.4: Perpendicular Line Tracking Starting from a New Configuration in K-region.

C. PLANNING STRATEGIES

The regions on the global path class provide information for rough navigation. The safe motion for accomplishing the mission will not be ensured without further elaborative planning. In order to simplify the complex task in mid-portion motion planning, we propose to solve the problem region by region starting from the first one of the mid-portion. That is to perform mid-portion motion planning in a single K-region independent of the situation in the neighboring K-regions. This strategy is possible as long as the motion in each individual region can be linked by continuous motion of its neighboring region.

To support this strategy, one possible method is to find a configuration on entrance and exit borders of a region and connect those configurations with a smooth path. The preprocessing step of mid-portion motion planning determined where to cross the two borders prior the main planning starts. Those crossing points together with their corresponding border orientations are used to define border configurations individually. Because the exit border of a region is exactly the entrance border of the region that follows it in the global path class, planning the motion between two configurations of the borders links the motions in two consecutive regions together. When the motions in all regions of the mid-portion of global path class are planned, a safe and smooth path is linked.

D. TYPES OF COMBINED MOTIONS

Three types of combined motions using the steering function are identified as useful in mid-portion motion planning. They are:

- Double parallel-line tracking.
- Double perpendicular-line tracking
- A parallel-line tracking followed by a perpendicular-line tracking, or vice versa.

The basic type of “double parallel-line tracking” motion is to track a line which is parallel to the Entrance configuration when the robot crosses the entrance border of the region. Then at certain positions it starts to track another line parallel to the previous one. Figure 4.5 illustrates the basic double parallel-line tracking motion. Since the reference lines will be computed according to the different region situations, this type of motion may have some variations. For instance, two reference lines can be the same line. Figure 4.11 shows this variation. Some other variations can be found in Figures 4.13, Figure 4.14, 4.17, 4.18, and 4.19.

The basic type of “double perpendicular-line tracking” motion is to track a line perpendicular to the Entrance configuration at the beginning and then at certain positions it start to track another line which is perpendicular to the current one. Figure 4.6 illustrates

the basic two perpendicular line tracking motion. The variations can be found in Figure 4.24 and Figure 4.25.

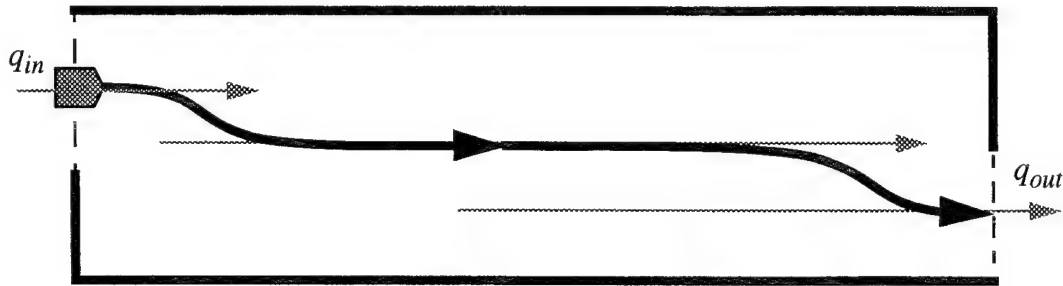


Figure 4.5: The Basic Double Parallel-Line Tracking Motion

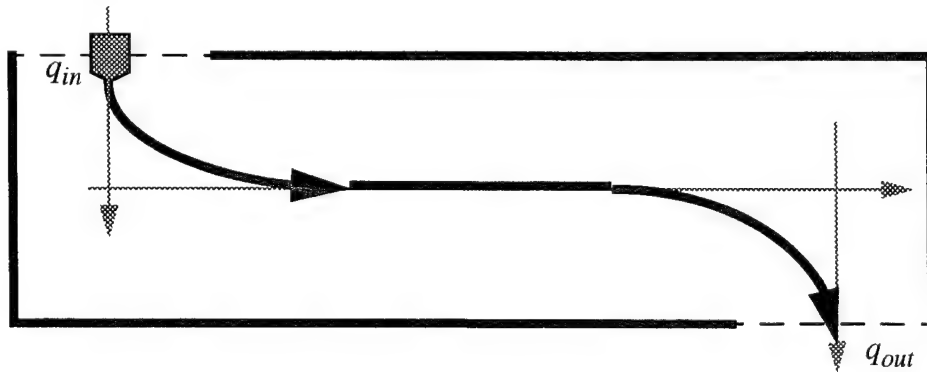


Figure 4.6: The Basic Double Perpendicular-Line Tracking

Another type of motion is to track a line parallel to the Entrance configuration, followed by tracking a line perpendicular to the current one, or vice versa. Figure 4.7 illustrates this basic one parallel line tracking followed by another perpendicular line tracking motion. As two parallel line tracking, this type of motion have some variations. The Figure 4.20, 4.21, 4.22, and 4.23 are the examples of the variations.

Recall that there are three major types on the positioning of the entrance and exit borders to the K-region as stated in Chapter III. What and how the basic motions is applied to planning motions in those distinct types of K-regions will be discussed in detailed in the following sections.

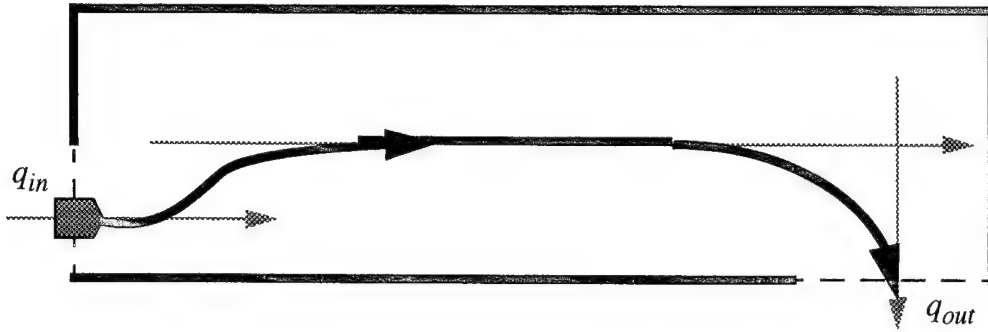


Figure 4.7: The Basic Motion of a Parallel Line Tracking Followed by a Perpendicular Line Tracking

E. PLANNING INTRA-REGION MOTION OF TYPE I

Obviously, this type of region has two borders with the same orientations, which means $\Psi_{in} = \Psi_{out}$. In a region of motion Type I, the basic motion is a forward motion of tracking a reference line which has orientation the same as entrance / exit border. This is under the condition that the reference line is long enough for line tracking motion to converge. Otherwise, different line tracking motion will be planned. Some factors should be considered in planning a safe motion in this type of region. They are the dimension of the region, the type of borders, and the position of borders in their corresponding region edge. Therefore we can further divide this type of region into three different categories as (i) Region with F-Borders in both entrance and exit borders as Figure 4.8. (ii). Region with one F-Border and one P-Border in entrance and exit as Figure 4.9. (iii). Region with P-Borders in both entrance and exit border as Figure 4.10.



Figure 4.8: The Category of K-region with F-Borders in Both Entrance and Exit Borders

1. Region with Two Full-Borders

This is the most common region we will meet in K-region decomposition. Since a K-region is a rectangle under our assumption, and since both entrance and exit border are Full-Border, it suggests that the crossing points of borders be the centers of borders. Thus the lines specified by entrance configuration $q_{in} = (BC_{in}, \Psi_{in}, 0)$ and exit configuration $q_{out} = (BC_{out}, \Psi_{out}, 0)$ are colinear with Horizontal Center Line.

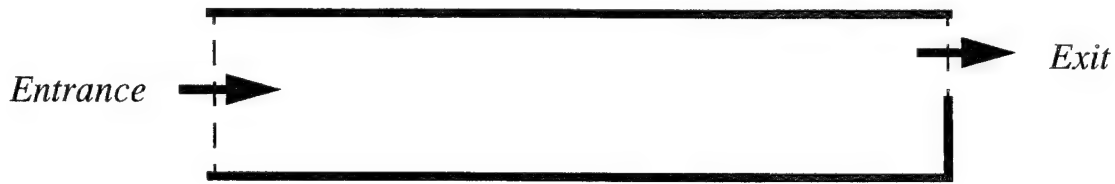


Figure 4.9: The Category of K-region with One F-Border and One P-Border in Entrance and Exit Border

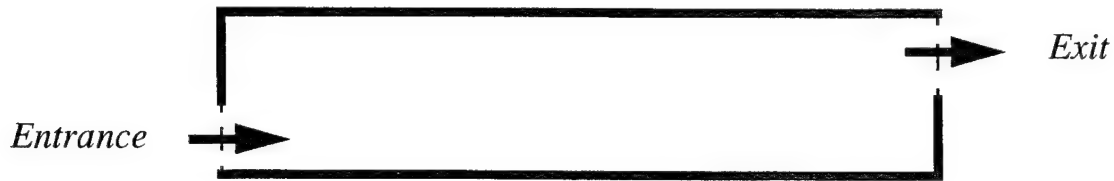


Figure 4.10: The Category of K-region with P-borders in Entrance and Exit Border Both

The center line of a region is the most desirable path in the sense of safety. Thus the motion in this kind of region is a forward motion of tracking the line specified by q_{out} from center of entrance border. Figure 4.11 illustrates an example of this motion.

2. Region with One Full-Border and One Partial-Border

The typical example of this kind of region is illustrated in Figure 4.12. In the figure, entrance and exit borders define two lines, which are $q_{in} = (P_{in}, \Psi_{in}, 0)$ and $q_{out} = (BC_{out},$

$\Psi_{out}, 0$). The crossing point of entrance border can be the point other than the center of borders if forward distance $FL < 2.02 * d_H$. There are two possible cases:

- The lines specified by q_{in} and q_{out} are colinear.
- The lines specified by q_{in} and q_{out} are not colinear.

The motion planning in these two cases will be discussed in the following subsections.

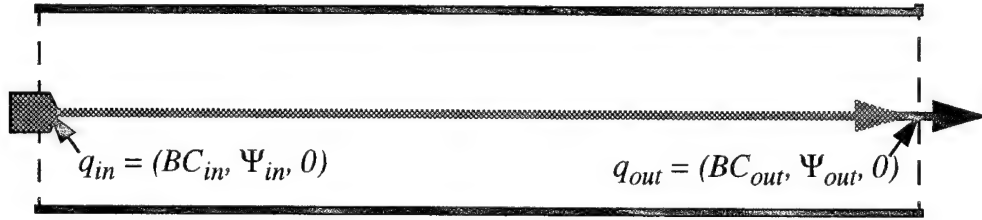


Figure 4.11: An Example of Forward Line Tracking Motion

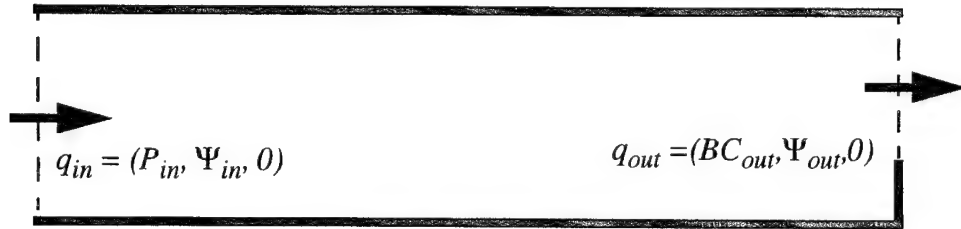


Figure 4.12: A Typical Region with One F-border and One P-border as Entrance and Exit Border.

a. Colinear Entrance and Exit Configurations

In this case, the exit configuration q_{out} must be defined by the center of the exit border and the Entrance configuration q_{in} can be defined by any point on the entrance border. Because the crossing point of the entrance border is predetermined as described in Chapter III, if it is not the center of the border, it implies tracking center line of the region

is not possible. Thus no matter what the q_{in} is, the motion will be a forward motion tracking the line specified by the configuration $q_{out} = (BC_{out}, \Psi_{out}, 0)$ with smoothness $\sigma = d_V / 11.22$ as illustrated in Figure 4.13.

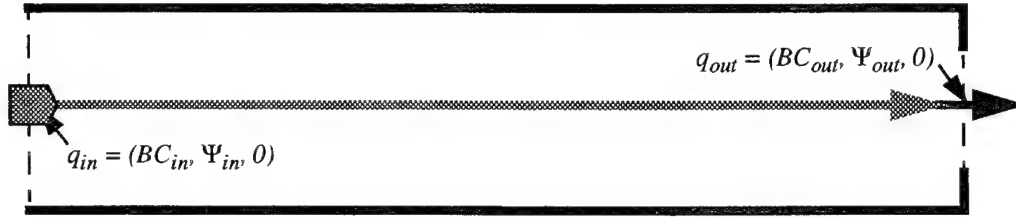


Figure 4.13: Forward Motion Tracking the Line q_{out}

b. Non-Colinear Entrance and Exit Configurations

In the region which has non-colinear entrance and exit configurations, the crossing point of the borders must be the centers of the borders and tracking center line is possible. This is how we determine the crossing point in this kind of special region. As the example illustrated in Figure 4.12, the entrance border is the F-Border. Thus the entrance configuration $q_{in} = (P_{in}, \Psi_{in}, 0)$ is actually specifying the center line of the region. The most straightforward motion in this region is to track the line specified by q_{in} at the beginning of entering the region, then at certain position leave the first line and start to track the parallel line specified by $q_{out} = (BC_{out}, \Psi_{out}, 0)$. However, as discussed in Chapter III, the trajectory of directly tracking two lines will not be symmetric to the trajectory traveled by the motion of reverse direction. Thus, planning motion in this type of region will need a reverse path C_{rev} as described in Chapter III to support the symmetric motion planning.

In this case, the reference line will be the Horizontal Center Line and $d_{Hin} = 0$ so that we have $d_H = d_{Hout}$. The *allowed convergence length* will be $L = d_V$ which is the length of the Horizontal Center Line. The smoothness can be computed as $\sigma_{in} = \sigma_{out} =$

$d_V / 11.22$ (see Eq 4.6). Then the reverse path C_{rev} is generated by tracking the Horizontal Center Line $q_c = (BC_{in}, \Psi_{in} + \pi, 0)$ from the configuration $q_s = (BC_{out}, \Psi_{out} + \pi, 0)$ with smoothness σ_{out} .

Motion in this region is executed by tracking the center line after entering the region followed by tracking the reverse path C_{rev} as Figure 4.14. Because the reverse path C_{rev} is a sequence of configurations and its first configuration q_n is exactly specifying a line the same as the configuration q_{in} specifies. Therefore, the entire motion in this region can be simplified by tracking the reverse path C_{rev} .

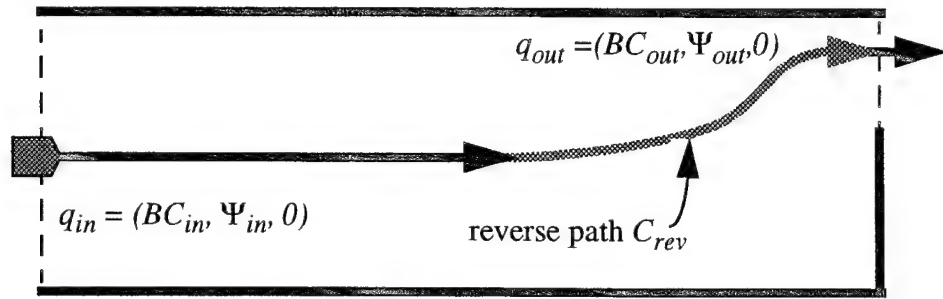


Figure 4.14: The Motion Planning in Region with One F-border and One P-border as Entrance and Exit.

For the case that the entrance border is a P-border and the exit border is an F-border, the motion in this region is simply to track the line specified by q_{out} from the configuration q_{in} . No reverse path is needed. Its trajectory will be exactly the same as the trajectory illustrated in Figure 4.14 except the motion orientation is opposite. Therefore the reverse path is needed only when the center of exit border does not coincide with its corresponding edge center in this kind of region.

3. Region with Two Partial-borders

There could be three situations in the region with P-borders in both entrance and exit borders. (i). Both centers of entrance and exit borders are aligned with centers of their

corresponding edges as shown in Figure 4.15(a). (ii). Forward length of the region $FL = d_V < 2.02 * d_H$ as shown in Figure 4.13(b). (iii). Forward length of the region $FL = d_V \geq 2.02 * d_H$ as shown in Figure 4.13(c). In the case of (i), the Local motion planning will be the same as the category in subsection 1 of this section. The cases (ii) and (iii) will be described in following subsections.

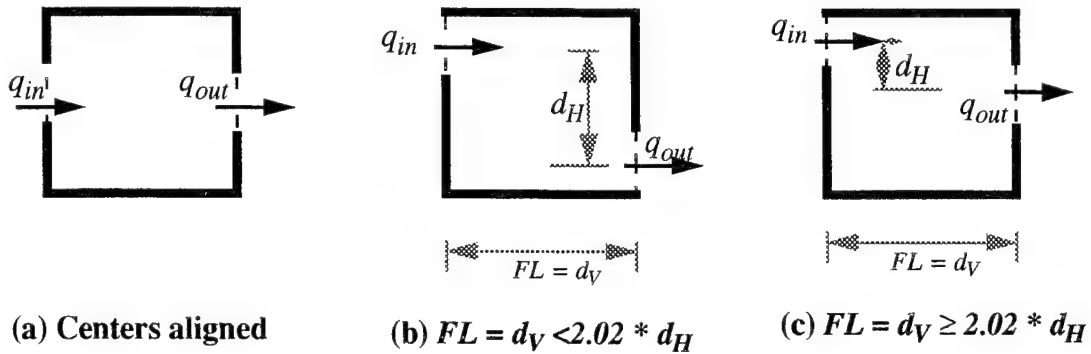


Figure 4.15: The Cases of the Region with P-border in Both Entrance and Exit Border

a. Forward Length $FL = d_V < 2.02 * d_H$

As described previously, the minimum length of reference line which allows parallel line tracking to converge is $L_{allowed} \geq 2.02 * d_{init}$. In our case as Figure 4.15 b, $L_{allowed} = d_V$ and $d_{init} = d_H$. Therefore, in this case since $d_V < 2.02 * d_H$, it is not possible to track a parallel reference line and converge to the line in the region. The motion planning in this case will be taking the Vertical Center Lined as reference line and perform two perpendicular line trackings. One from entrance configuration q_{in} and another one from Exit configuration q_{out} . This implies $d_V = d_{V_{in}} + d_{V_{out}}$ and $d_{V_{in}} = d_{V_{out}}$. The later perpendicular line tracking will be performed by generating a reverse path to follow so that the motion is symmetric. In order to ensure perpendicular line tracking convergence, Eq

4.10 is adopted. Because there will be two perpendicular line tracking with same reference line, the length of reference line that allows line tracking to converge will be $d_H \geq 2 * 3.38 * d_V$. If this is satisfied, then the perpendicular line tracking can be performed from the configurations q_{in} and q_{out} as Figure 4.16(a). Otherwise the start configurations need to be computed. The new start configurations, q_{sin} and q_{sout} are on the line q_{in} and q_{out} such that $d_{Vnew} = d_H / (2 * 3.38)$ where d_{Vnew} is the distance as shown in Figure 4.16(b). The smoothness for one perpendicular line trackings can be calculated as Eq 4.4. In the case of double perpendicular line tracking on the same reference line, we have:

$$\sigma_{in} = \sigma_{out} = 0.42 * d_V$$

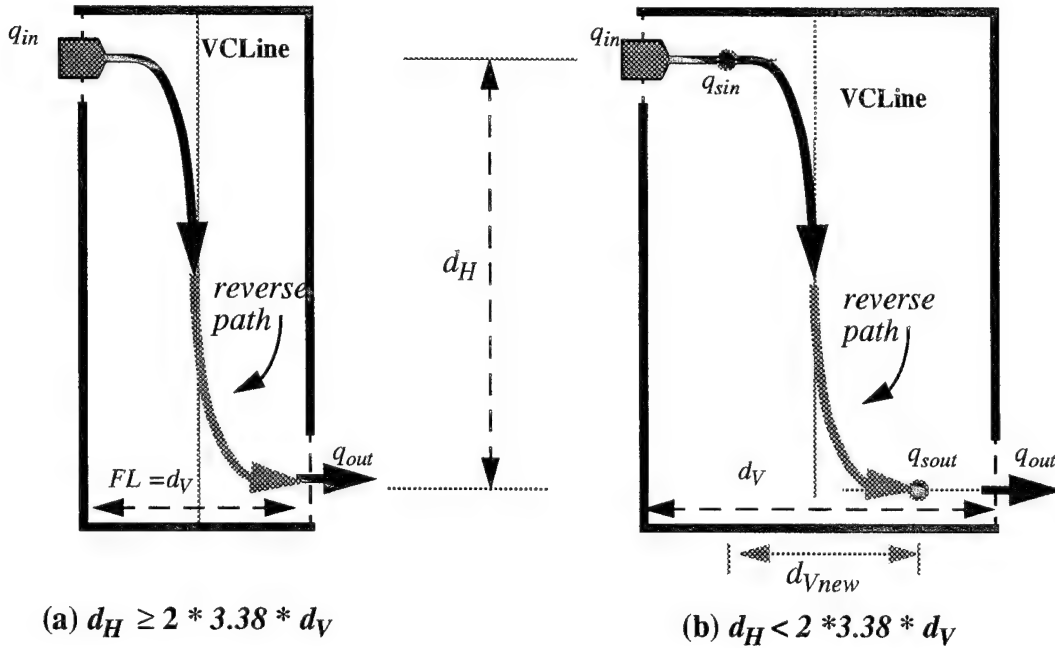


Figure 4.16: The Example of Double Perpendicular Line Tracking

The motion planning for the region in this case is as follows:

- Motion Type: perpendicular line tracking:
- Reference line: Vertical Center Line
- Compute perpendicular line tracking start configuration q_{sin} and q_{sout} on line

q_{in} , q_{out} respectively.

- Compute perpendicular line tracking smoothness σ_{in} and σ_{out}
- Generate reverse path C_{rev} by tracking q_c from q_{sout}
- The motion execution will be: a. Track the line q_{in} from configuration q_{in} until start configuration q_{sin} is reached. b. Track the reverse path C_{rev} .

b. Forward Length $FL = d_V \geq 2.02 * d_H$

In the region with the forward length $FL = d_V \geq 2.02 * d_H$, a parallel line tracking is expected. Ideally, tracking the Horizontal Center Line of this type of the region will be the most desirable motion. However, taking the center line as reference line is not always possible for many situations. We discuss this in following three different situations: (i). One of crossing point of Entrance or exit borders coincides with the center of its corresponding edge. (ii). Both crossing point of entrance and exit borders, are on the same side of the Horizontal Center Line. (iii). The crossing point of entrance and exit borders, are on the opposite side of the Horizontal Center Line of the region.

For the case (i), the local motion planning will be similar to the motion planning in the region with one F-border and one P-border in entrance and exit border described earlier in this chapter. Keep in mind that the reverse path is generated only if the crossing point of exit border does not coincide with its corresponding edge center. The followings are the motion planning in the rest of cases.

(1) Crossing Points on the Same Side of Center Line: Because the crossing points of two borders are on the same side of the center line, to track the center line must take some extra efforts. Thus whether to track the center line is an important decision. Since the safe motion is our major concern in local motion planning, and tracking the center line in a region is considered the safest motion, our decision for this is to track the center line whenever it is possible. As analysis in this chapter earlier, the converge length for a parallel line tracking is $2.02 * d_{init}$. Therefore if the Eq 4.11 is satisfied, the (parallel) center line tracking motion will be performed:

$$d_V \geq 2.02 (d_{Hin} + d_{Hout}) \quad (\text{Eq 4.11})$$

where d_{Hin} and d_{Hout} are the distances from Horizontal Center Line to q_{in} and q_{out} respectively. In the center line tracking motion planning, a reverse path C_{rev} is always generated from the crossing point of exit border reversely. Because distances d_{Hin} and d_{Hout} may be different, the smoothness σ_{in} and σ_{out} need to be computed separately as follows:

$$\sigma_{in} = \frac{FL \times d_{Hin}}{11.22 \times (d_{Hin} + d_{Hout})} \quad (\text{Eq 4.12})$$

$$\sigma_{out} = \frac{FL \times d_{Hout}}{11.22 \times (d_{Hin} + d_{Hout})} \quad (\text{Eq 4.13})$$

The entire motion in this region then is tracking the reverse path C_{rev} as Figure 4.17.

On the other hand, if $FL = d_V < 2.02 * (d_{Hin} + d_{Hout})$, the forward length of the region is not long enough for tracking Horizontal Center Line and converging on the line. In this case, instead of tracking the center line, the reference line will be chosen according the distance from the crossing point of borders to the Horizontal Center Line. Since the motion which is closer to the center line is consider safer, we take the line which is closer to center line as the reference line. The smoothness is computed as Eq 4.14

$$\sigma_{in} = \sigma_{out} = d_V / 11.22 \quad (\text{Eq 4.14})$$

For symmetry reason, when the reference line is q_{in} , a reverse path C_{rev} will be generated from q_{out} with reverse orientation. The entire motion in the region of this situation is tracking the reference line or the reverse path C_{rev} if it exists. The Figure 4.18 illustrates the example.

(2) Crossing Points on the Opposite Side of Center Line: In the region of crossing points on the opposite side of the Horizontal Center Line, because the center line is between q_{in} and q_{out} , and the length of the center line is long enough for parallel line tracking to converge, the motion in this region will be simply tracking the center line as

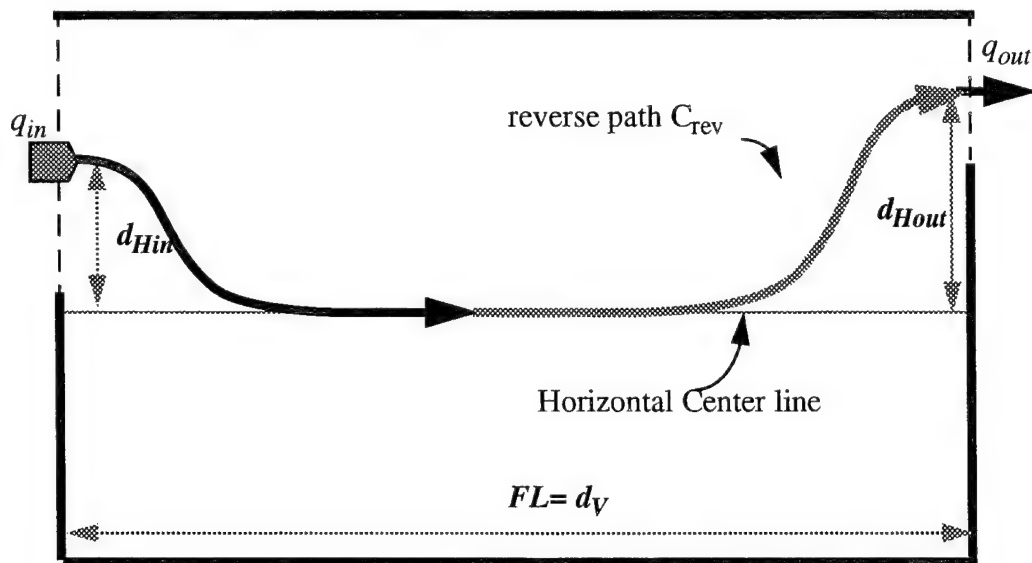


Figure 4.17: The Example of the Motion Tracking the Horizontal Center Line When $FL = d_v \geq 2.02 * (d_{Hin} + d_{Hout})$.

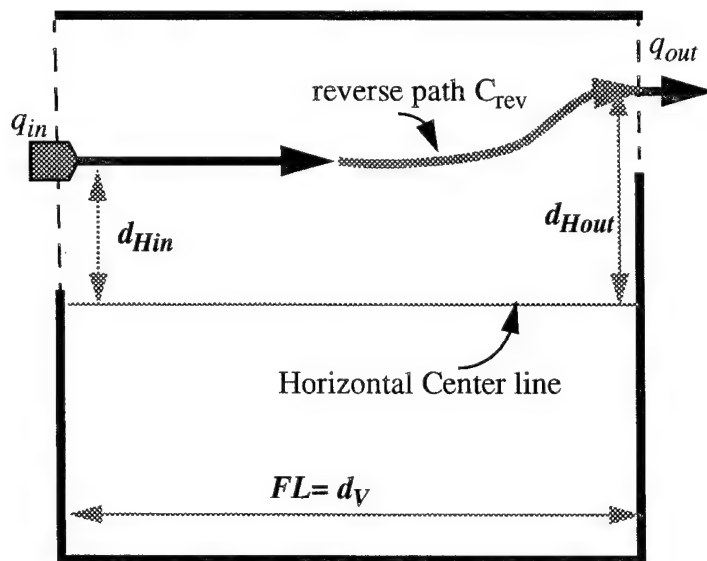


Figure 4.18: The Example of the Motion Tracking a Reference Line Closer to the Horizontal Center Line When $FL = d_v < 2.02 * (d_{Hin} + d_{Hout})$.

illustrated in Figure 4.19. The computation of smoothness and motion planning for this situation is similar to that of the first case of the region with crossing points on the same side of the Horizontal Center Line.

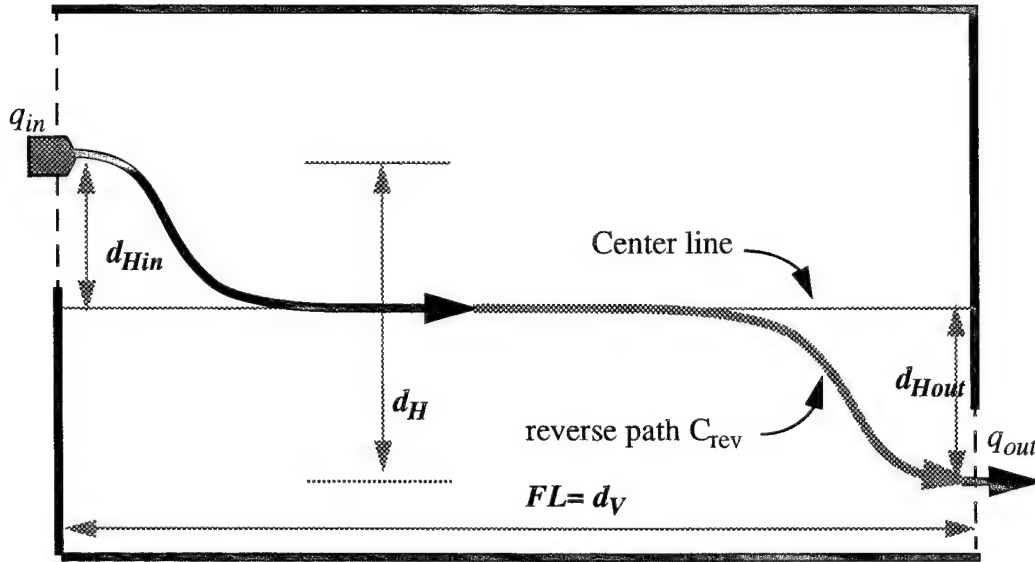


Figure 4.19: The Example of the Motion Planning in the Region with q_{in} and q_{out} on the Opposite Side of the Horizontal Center Line

F. PLANNING INTRA-REGION MOTION OF TYPE II

The characteristics of this type of region is that entrance orientation is perpendicular to the orientation of exit border. Therefore the basic motion in this type of region is a left or right turning motion.

There are basically two categories in this type of region which is separated by the relationship between Forward Length FL and Cross Length CL of the region.

1. Forward Length Greater than or Equal to Cross Length

In the region of $FL \geq CL$, the main reference line will be the Horizontal Center Line defined by bisector two Forward Edges. Since the motion tracking the center line is always considered a safe motion, we will plan the detailed motion which follows the center as

much as possible. However, as in other type of region, it is not always possible to track the center because of its length limitation. If tracking the center line is taken, the motion must be a parallel line tracking at the beginning of entering the region and followed by a perpendicular line tracking to cross the exit border. In order to plan a symmetric path, a reverse path will be generated with perpendicular line tracking starting at exit border. Thus, both line trackings take the center line as reference line. This give us a hint in calculating how long the center line is needed for parallel and perpendicular line tracking to converge. As analyzed in this chapter earlier, the minimum length on reference line that allows parallel line tracking to converge is $2.02 * d_{init}$ and in this case we have $d_{init} = d_{Hin}$. The minimum length on reference line that allows perpendicular line tracking to converge is $3.38 * d_{init}$. In this case we have $d_{init} = d_{Hout}$. Because the center line is parallel to the line specified by q_{in} , the distance d_H is total length on center line that allows both line trackings to converge. Therefore, whether to track the center line in this kind of region depends whether the following inequality is satisfied.:

$$d_V \geq (2.02 * d_{Hin} + 3.38 * d_{Hout}) \quad (\text{Eq 4.15})$$

When Eq 4.15 is satisfied, tracking the center line is possible. A reverse path C_{rev} will be generated after their individual smoothness is calculated as Eq4.16 and 4.17.

$$\sigma_{in} = \frac{d_V - 3.38 \times d_{Hout}}{11.22} \quad (\text{Eq 4.16})$$

$$\sigma_{out} = 0.42 \times d_{Hout} \quad (\text{Eq 4.17})$$

Then the motion in this region is to track the reverse path C_{rev} as Figure 4.20 and Figure 4.21.

If Eq 4.15 is not satisfied, center line tracking will not be possible. Then a single perpendicular line tracking motion will be planned. Either the line specified by entrance configuration or the line specified by exit configuration will be chosen as the reference line depending on the length it allows line tracking to converge. For consistence, if the distance $d_V \geq d_H$, which means the line specified by q_{in} will be longer (measured from the crossing

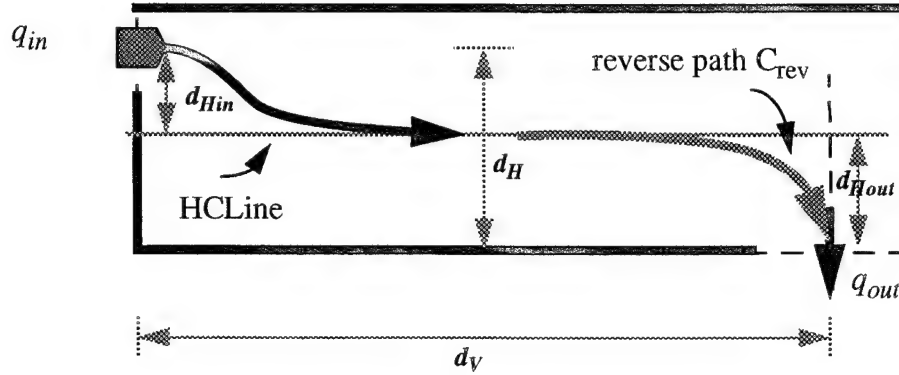


Figure 4.20: The Example I of Tracking Horizontal Center Line in the Region of Type II.

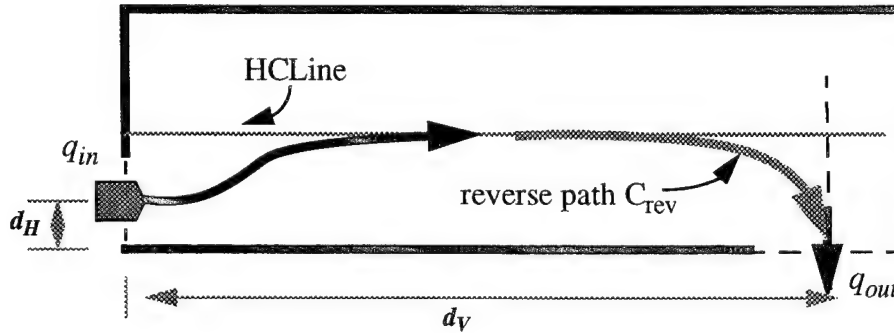


Figure 4.21: The Example II of Tracking Horizontal Center Line in the Region of Type II.

point to the intersection of two lines q_{in} and q_{out}), then the line q_{in} will be the reference line. Again for the symmetric path consideration, a reverse path may be generated if the reference line is entrance configuration q_{in} . With the reference line selected, we are able to compute the start configuration q_{sout} such that its distance to reference line satisfies $d = d_V / 3.38$. Then its smoothness for perpendicular line tracking will be $\sigma = 0.42 * d$. We generate the reverse path C_{rev} by tracking the reference line from q_{sout} with reverse direction. The entire motion in this region then is tracking the reverse path C_{rev} . Figure

4.22 illustrates the single perpendicular line tracking planning. If $d_V < d_H$ then the line q_{out} will be selected as reference line. The start configuration q_{sin} and its smoothness σ for perpendicular line tracking can be calculated in similar way. The motion in this region then is planned to track q_{in} when entering the region until reaches q_{sin} . Then perform perpendicular line tracking with q_{out} as reference line. No reverse path is needed in this case.

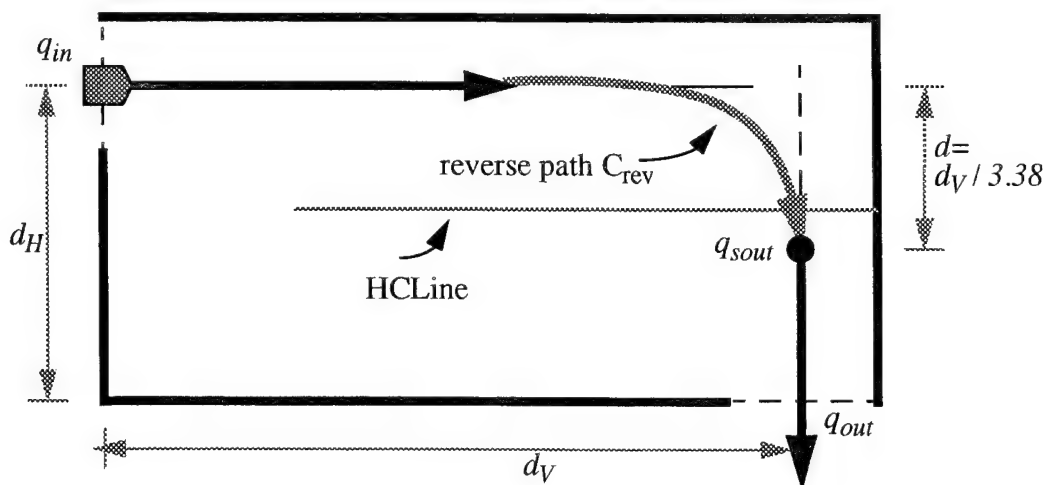


Figure 4.22: The Example of Single Perpendicular Line Tracking in the Region of Type II

2. Forward Length Smaller than Cross Length

In the region of Forward Length less than Cross Length, the Vertical Center Line of the region define by the bisector of two Cross Edges is to be considered as reference line if possible. The local motion planning rule in that kind of region is similar the region of $FL \geq CL$ except the center line is defined by different pair of edges. Figure 4.23 illustrates an example of the motion planning in this region.

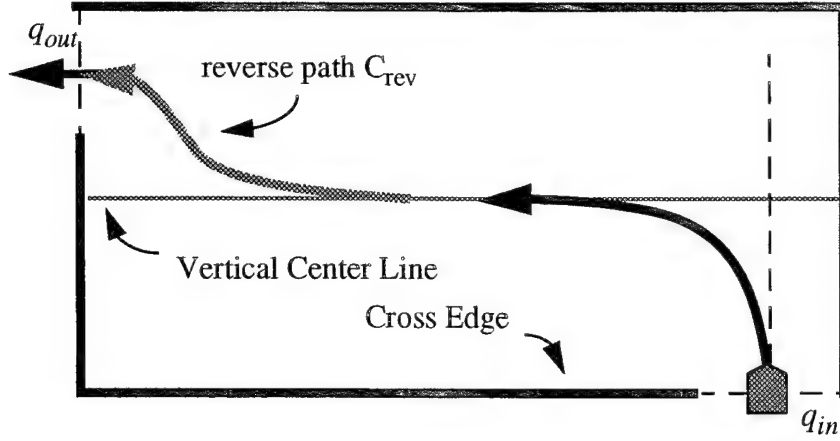


Figure 4.23: An Example of Tracking Center Line Defined by Two Cross Edges of the Region of Type II

G. PLANNING INTRA-REGION MOTION OF TYPE III

In the region of the entrance and exit border on the same edge, the motion can be planned with the combination of two perpendicular line trackings. The reference line will be the Vertical Center Line or a line parallel to it. In this type of region, we will consider whether to track the Vertical Center Line or not. Let d_H , d_{vin} and d_{vout} be the distances as shown on the Figure 4.24. For a perpendicular line tracking, we have the restriction that the length on the reference line must be $L_{allowed} \geq 3.38 * d_{init}$ (Eq 4.10). Thus, if Eq 4.18 is satisfied, then the Vertical Center Line will be the reference line of the perpendicular line trackings.

$$d_H \geq (3.38 * d_{vin} + 3.38 * d_{vout})$$

$$d_H \geq 2 * 3.38 * d_{vout} \quad (\text{Eq 4.18})$$

And the smoothness for the line tracking is

$$\sigma_{in} = \sigma_{out} = 0.42 * d_{vout}$$

For symmetric motion, the reverse path C_{rev} is generated by tracking the center line from the exit configuration q_{out} with reverse orientations. Then the entire motion in this region is tracking the reverse path C_{rev} . The motion of tracking center line of this region is illustrated in Figure 4.24.

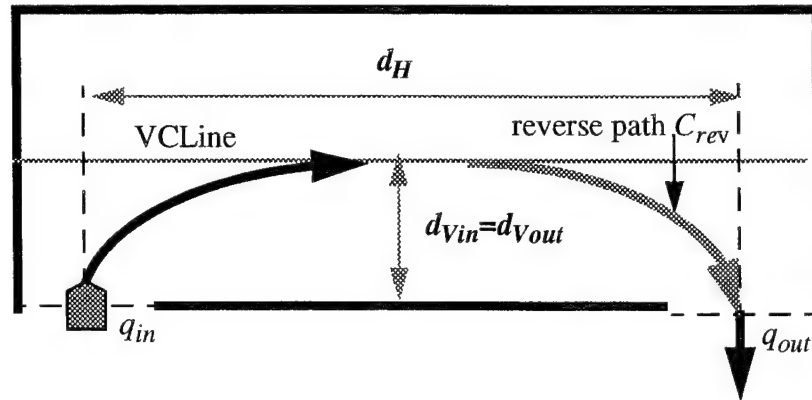


Figure 4.24: The Motion of Vertical Center Line Tracking in the Region of Motion Type III.

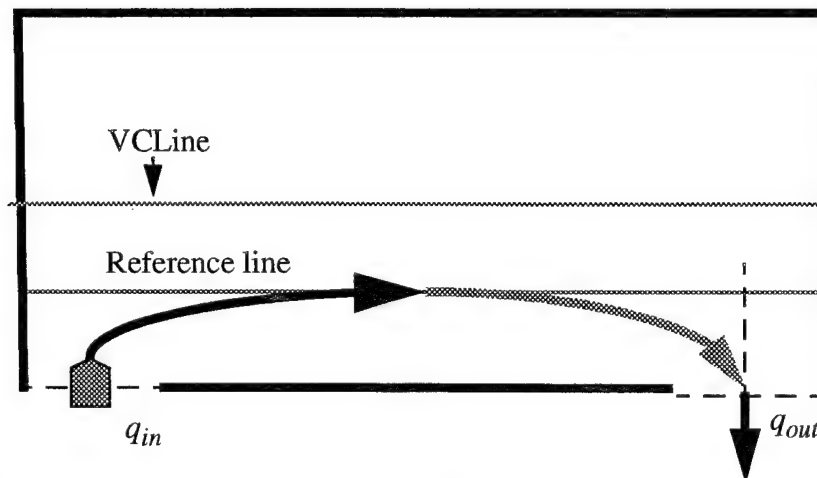


Figure 4.25: The Motion of a Reference Line Tracking in the Region of Motion Type III

For the case that $d_H < 2 * 3.38 * d_{Vout}$, instead of tracking the Vertical Center Line, the motion is tracking a reference line parallel to the center line such that $d_H = 2 * 3.38 * d_{Vout}$ where d_{Vout} is the distance between the reference line to the edge containing the borders. The rest of planning in this region will be all the same. Figure 4.25 illustrates this type of motion planning.

H. MID-PORTION MOTION PLANNING RULES

The previous sections analyzed how the local motion planning in the individual regions is done. We summarize those analysis into motion rules based on the type of regions. Each of different type of regions can be subdivided into two planning situations depending on the region feature which allows certain type of line tracking being performed. Figure 4.26 shows the decision tree for the mid-portion motion planning rules.

1. Region with Two Parallel Borders

$$a. \quad d_V \geq 2.02 (d_{Hin} + d_{Hout})$$

(1) Motion type: Two parallel line trackings.

(2) Reference line: Horizontal Center Line.

(3) Planning:

(a) Compute smoothness:

For σ_{in} , if $d_{Hin} = 0$, $\sigma_{in} = d_V / 11.22$, else Eq 4.2

For σ_{out} , if $d_{Hout} = 0$, $\sigma_{out} = d_V / 11.22$, else Eq 4.3

(b) Generate a reverse path C_{rev} from q_{sout} using smoothness σ_{out} .

(4) Motion execution:

Tracking path C_{rev} with smoothness σ_{in} .

(5) Example:

Figure 4.11, Figure 4.13, Figure 4.14, Figure 4.17, Figure 4.19.

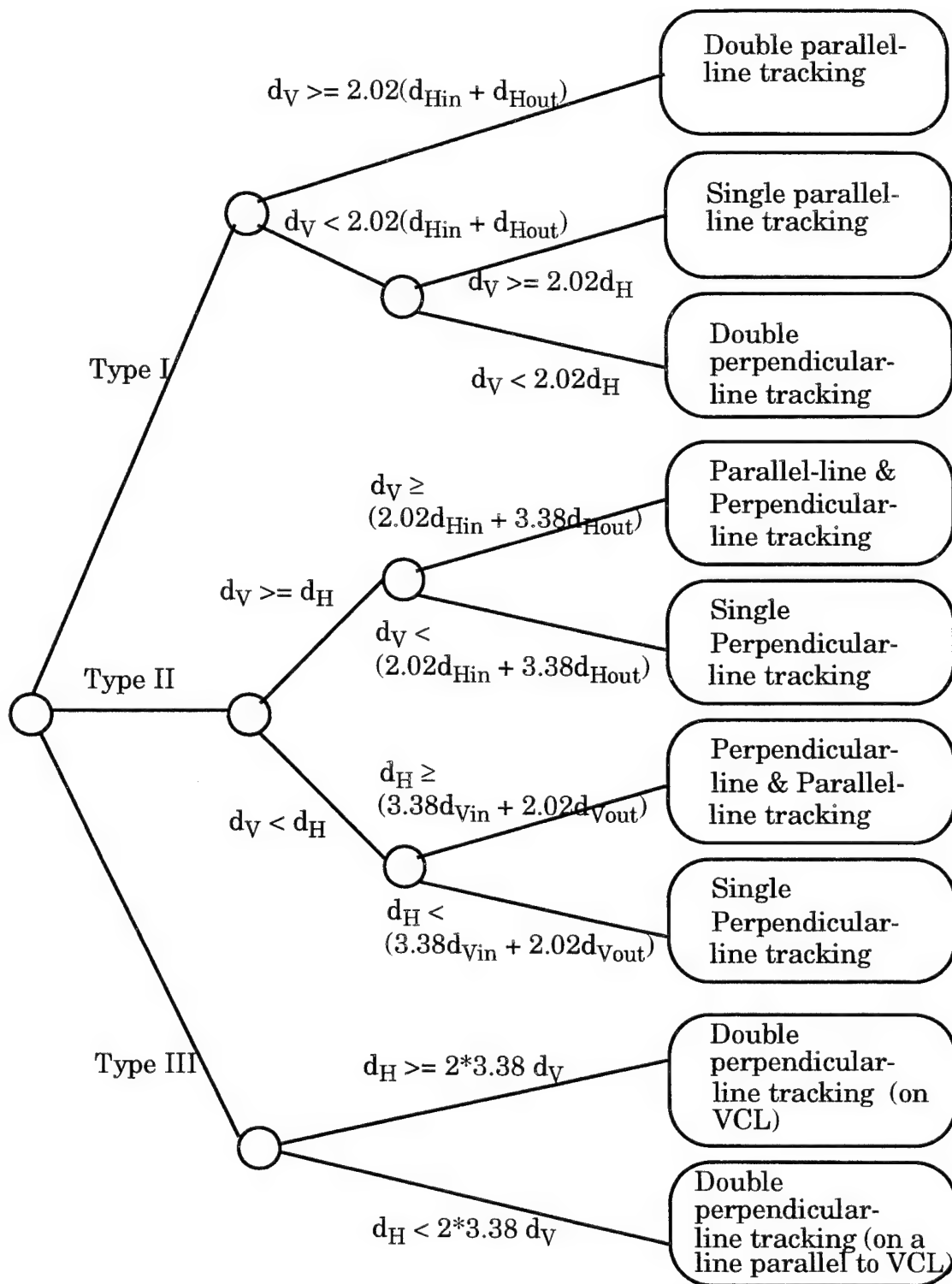


Figure 4.26: The Decision Tree for Mid-portion Motion Planning Rules

b. $d_V < 2.02 (d_{Hin} + d_{Hout})$

Case 1: $d_V \geq 2.02 d_H$

(1) Motion type: Single parallel-line trackings.

(2) Reference line:

$$q_{in} \quad \text{if } d_{Hin} < d_{Hout};$$

$$q_{out} \quad \text{Otherwise.}$$

(3) Planning:

(a) Compute smoothness $\sigma = d_V / 11.22$.

(b) if $d_{Hin} < d_{Hout}$, generate reverse path C_{rev} from q_{out} .

(4) Motion execution:

if $d_{Hin} < d_{Hout}$, tracking reverse path C_{rev} , otherwise tracking q_{out} .

(5) Example:

Figure 4.18.

Case 2: $d_V < 2.02 d_H$

(1) Motion type: Double perpendicular-line trackings.

(2) Reference line: Vertical Center Line.

(3) Planning:

(a) Compute start configurations q_{sin} and q_{sout} for perpendicular-line tracking.

(b) Compute smoothness $\sigma = 0.42 d_V$.

(c) Generate reverse path C_{rev} from q_{sout} .

(4) Motion execution:

(a) Tracking entrance line q_{in} until the configuration q_{sin} .

(b) Tracking the reverse path C_{rev} with smoothness σ .

(5) Example:

Figure 4.16.

2. Region with Two Perpendicular Borders

a. $d_V \geq d_H$

(1) Motion type: Parallel-line tracking for entrance motion

Perpendicular-line tracking for exit motion

(2) Reference line:

Horizontal Center Line if $d_V \geq (2.02 d_{Hin} + 3.38 d_{Hout})$,

q_{in} Otherwise.

(3) Planning:

(a) If reference line is q_{in} , compute start configurations q_{sout} and its distance to the reference line, d_{Hout} , for perpendicular-line tracking as illustrated in Figure 4.22.

(b) Compute smoothness:

$$\sigma_{in} = (d_V - 3.38 d_{Hout}) / 11.22, \sigma_{out} = 0.42 d_{Hout}$$

if $\sigma_{in} \leq 0$, $\sigma_{in} = \sigma_{out}$.

(c) Generate a reverse path C_{rev} from q_{sout} using smoothness σ_{out} tracking reversed reference line.

(4) Motion execution:

Tracking reverse path C_{rev} with smoothness σ_{in} .

(5) Example:

Figure 4.20, Figure 4.21, Figure 4.22.

b. $d_V < d_H$

(1) Motion type: Perpendicular-line tracking for entrance motion

Parallel-line tracking for exit motion.

(2) Reference line:

Vertical Center Line if $d_H \geq (3.38 d_{Vin} + 2.02 d_{Vout})$,

q_{out} Otherwise.

(3) Planning:

(a) If reference line is q_{out} , compute start configurations q_{sin} and d_{vin} for perpendicular-line tracking similar to Figure 4.22.

(b) Compute smoothness:

$$\sigma_{in} = 0.42 d_{vin}, \sigma_{out} = (d_H - 3.38 d_{vin}) / 11.22.$$

$$\text{if } \sigma_{out} \leq 0, \sigma_{out} = \sigma_{in}.$$

(c) Generate a reverse path C_{rev} from q_{out} using smoothness σ_{out} tracking reversed reference line.

(4) Motion execution:

(a) Tracking entrance line q_{in} until the configuration q_{sin} .

(b) Tracking the reverse path C_{rev} with smoothness σ_{in} .

(5) Example:

Figure 4.23.

3. Region with Borders on the Same Edge

a. $d_H \geq 2 * 3.38 d_V$

(1) Motion type: Double perpendicular-line trackings.

(2) Reference line: Vertical Center Line.

(3) Planning:

(a) Compute smoothness $\sigma = 0.42 d_V$.

(b) Generate reverse path C_{rev} from q_{out} with smoothness σ .

(4) Motion execution:

Tracking the reverse path C_{rev} with smoothness σ .

(5) Example:

Figure 4.24.

b. $d_H < 2 * 3.38 d_V$

(1) Motion type: Double perpendicular-line trackings.

(2) Reference line:

A line or parallel to Vertical Center Line such that its distance to the border $d_r = d_H / (2 * 3.38)$

(3) Planning:

(a) Compute smoothness $\sigma = 0.42 d_r$.

(b) Generate reverse path C_{rev} from q_{out} with smoothness σ .

(4) Motion execution:

Tracking the reverse path C_{rev} with smoothness σ .

(5) Example:

Figure 4.25.

I. ALGORITHM

The mid-portion motion planning is performed region by region starting from the first region of global path class which is not included in initial motion planning, and ending in the region which is the last region on the path that is included in final motion planning. The following algorithm of mid-portion motion planning is designed for a single region on the mid-portion of global path class. Thus, the inputs of the algorithm are the entrance and exit configurations, the current region identification and the world model. The output of the algorithm is the motion instructions which includes a reference path, named reverse path represented by a sequence of configurations, a smoothness, which control the sharpness of turning while tracking a reference line, and a starting configuration which indicates where to start the path tracking. Based on the rule described above, the algorithm is developed as below.

Algorithm **MidPortionMP** (q_{in} , q_{out} , $r_{current}$, W)

Input: q_{in} : entrance configuration;

q_{out} : exit configuration;

$r_{current}$: current region;

W : a world model

Output: MP : motion instruction

```

(1)  region =  $r_{current}$ ;
(2)   $qIn = q_{in}$ ;
(3)   $qOut = q_{out}$ ;
(4)  ComputeDis (region,  $qIn$ ,  $qOut$ );
(5)  if (regionType(region) = Type1) then /* parallel borders */
(6)      if ( $dv \geq 2.02 * (d_{hin} + d_{hout})$ ) then /* Two parallel line tracking */
(7)           $refLine = HCL$  /* Horizontal Center Line */
(8)          ComputeSigma ( $\Sigma_{in}$ ,  $\Sigma_{out}$ , rule1);
(9)      else
(10)         if ( $dv \geq 2.02 * dh$ ) then /* single line tracking */
(11)             if ( $d_{hin} < d_{hout}$ ) then /* select the line closer to center line */
(12)                  $refLine = qIn$ 
(13)             else
(14)                  $refLine = qOut$ 
(15)             ComputeSigma ( $\Sigma_{in}$ ,  $\Sigma_{out}$ , rule2.1);
(16)         else /* two perpendicular line tracking */
(17)              $refLine = VCL$  /* vertical center line */
(18)             ComputeSigma ( $\Sigma_{in}$ ,  $\Sigma_{out}$ , rule2.2);
(19)             ComputeTrackConfig ( $refLine$ ,  $q_{in}$ ,  $q_{out}$ );
(20)     else if (regionType(region) = Type2) then /* perpendicular borders */
(21)         if ( $dh \geq dh$ ) then /* single line tracking */
(22)             if ( $dv \geq (2.02 * d_{hin} + 3.38 * d_{hout})$ ) then /* HCL tracking (H&V) */
(23)                  $refLine = HCL$ 
(24)             else
(25)                  $refLine = qIn$ 
(26)                 ComputeTrackConfig ( $refLine$ ,  $q_{in}$ ,  $q_{out}$ );
(27)                 ComputeSigma ( $\Sigma_{in}$ ,  $\Sigma_{out}$ , rule3);
(28)             else
(29)                 if ( $dh \geq (3.38 * d_{vin} + 2.02 * d_{vout})$ ) then /* VCL tracking (V&H) */
(30)                      $refLine = VCL$ 
(31)                 else /* tracking entrance line */
(32)                      $refLine = qOut$ 
(33)                     ComputeTrackCofig ( $refLine$ ,  $q_{in}$ ,  $q_{out}$ );
(34)                     ComputeSigma ( $\Sigma_{in}$ ,  $\Sigma_{out}$ , rule4);
(35)     else if (regionType(region) = Type3) then /* borders on same edge */

```



```

(36)   if ( $dh \geq 3.38 * (dvin + dvout)$ ) then /* Two perpendicular tracking */
(37)        $refLine = VCL$ 
(38)       ComputeSigma ( $SigmaIn, SigmaOut, rule5$ );
(39)   else /* tracking a new line parallel to VCL*/
(40)        $refLine = computeRefLine()$ ;
(41)       ComputeSigma ( $SigmaIn, SigmaOut, rule6$ );
(42)    $MP.rvsPath = genRvsPath (refLine, qOut, SigmaOut)$ 
(43)    $MP.sigma = SigmaIn$ ;
(44)    $MP.startConfig = qIn$ ;
(45)   return ( $MP$ );

```

The subroutine **ComputeDis** computes all distance measurements in a given region as defined in Section A. The measurements include dv , dh , $dvin$, $dvout$, $dhin$, and $dhout$, Subroutine **regionType** takes region name as input and returns the type of region which is Type1, Type2 and Type3 as described in Chapter III Section C. Subroutine **ComputeSigma** computes smoothness according to different region situations which are described in Section H. The subroutine **ComputeTrackCofig** computes the start configurations of path tracking if necessary. The situations that need to compute start configuration of path tracking are described in Section H. The subroutine **genRvsPath** generates a reverse path represented by a sequence of configurations according to given reference line, starting configuration, and smoothness. The reverse path generation uses forerunner simulation as tool. The details of reverse path generation are described in Section E and F of Chapter III.

V. END-PORION MOTION PLANNING

The end-portion motion planning refers to initial motion planning and final motion planning. Let q_s and q_g be the start and goal configurations respectively. Assume there is a collision-free motion planning from q_s to q_g , with path Π_1 . In the sense of symmetric motion planning we stress in this dissertation, if the orientations of all configurations are reversed, the path Π_2 of the motion planned from the final configuration to initial configuration will be identical to Π_1 . Therefore the planning for the initial motion would be similar to that of the final planning except they are reverse in orientations. The chapter addresses how the end-portion motion will be planned.

A. PROBLEM STATEMENT

Since the two plannings in end-portion motion planning are actually the mirror of each other, the problem to be solved in final motion planning is the same as that of initial motion planning. We describe the initial motion planning problem as the following, so that the final motion planning problem becomes trivial.

Given a world model with decomposed K -regions on a two dimensional plane, \mathbb{R}^2 , and a global path class $\Pi = \langle R_{i1}, B_{i1}, \dots, R_{in-1}, B_{in-1}, R_{in} \rangle$, with start configuration q_s and goal configuration q_g . The crossing points of borders are determined initially. The initial motion planning problem is to determine whether there exists a collision-free motion for a rigid body robot to move from q_s to the regions next to the initial one crossing the border perpendicularly. If so, plan a safe motion symmetrically.

B. PLANNING METHODS

The inputs to the end-portion motion planning are the initial mission parameters, i.e. start configuration, q_s , goal configuration q_g , and the world model, W , and the result of global path planning, i.e. the global path class Π . The output of this planning is a sequence of motions instructions, $MP_{end} = \langle MP_1, \dots, MP_n \rangle$, to be taken, where $n \geq 1$ and each MP_i

includes type of motion (either forward or backing-up), a reference path, the configurations to initialize and end the motion. Figure 5.1 illustrates this idea.

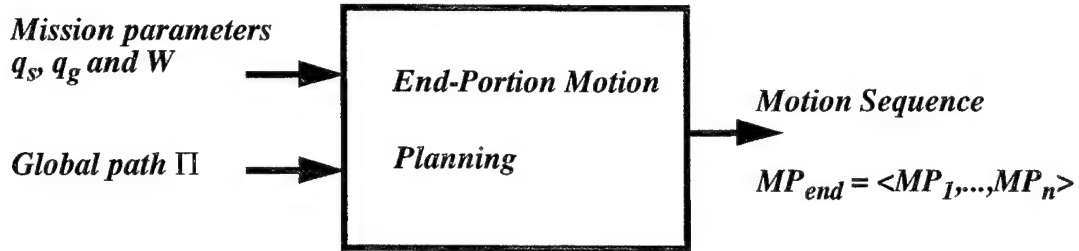


Figure 5.1: The End-Portion Motion Planning

According to these I/O requirements, the end-portion motion planning will be designed to use methods described in this section to accomplish the planning job. The Forerunner Simulation has the key role in the entire end-portion motion planning. How to apply the Forerunner Simulation to end-portion motion planning is described in Subsection 1. To plan a collision free motion is the main purpose of end-portion motion planning. Therefore, collision detection is one of important topics that will be discussed in Subsection 2. In the simulation, if no forward motion is possible, backing up motions will be planned as a supplemental motion. We discuss this in Subsection 3.

1. Forward Forerunner Simulation Application

The generic Forerunner Simulation template is presented in Chapter VI. In this chapter we discuss how it is applied to a forward simulation in the end-portion motion planning. The path tracking technique is one of good methods in solving local motion planning problem. Because the Forerunner Simulation is basically using line tracking method to simulate a real robot's motion, we found it is especially useful in end-portion motion planning.

Given an initial configuration $q_{int} = ((x_{int}, y_{int}), \theta_{int}, k_{int})$, a reference path specified by the configuration $path = ((x_p, y_p), \theta_p, k_p)$ and a smoothness σ , what we expect the Forerunner Simulation to do is to perform a forward line tracking with following output:

- If the forerunner's trajectory converges to the reference line before the point on the line where it is supposed to converge, output the line tracking trajectory from q_{int} to the configuration where it converges. Otherwise indicates not converging case.
- If collision happens, output a collision signal.

With these expectation, a forerunner simulator can be constructed by modifying the simulation template in Figure 3.6 of Chapter III. Figure 5.2 and 5.3 show the algorithms.

As aforementioned, the Forerunner Simulation can be applied to many different applications. In the ForwardSimulation application, the related codes are inserted to the template. In Figure 5.2, line (1), (2), and (3) are the PREwork and line (8) to (14) are the application codes. The collision checking function in line (8) will be discussed later in Subsection 3. The function `overImage()` in line (10) is to check the image [19] of current configuration on the reference line with the point where the trajectory is supposed to converge before it. If the image runs over the point, the function returns TRUE, otherwise FALSE is returned. The algorithm for the function `overImage()` is omitted. The convergence checking algorithm in line (11) is presented in Figure 5.3. The algorithm of ForwardSimulation returns a path Π_{fw} containing a sequence of configurations of the tracking trajectory. Two other boolean variables, `FWcollide` and `FWPerfectConverge`, are also updated and returned. `FWcollide` indicates whether the simulation collides with objects and `FWPerfectConverge` indicates whether the trajectory converges before the point specified in the reference line.

Algorithm ForwardSimulation (q, q_{fw}, σ, W)

Input: q : an initial configuration; q_{fw} : a reference path; σ : a smoothness;
 W : a world model;

Output: update forward path Π_{fw} ; update FWcollide and FWPerfectConverge;

- (1) Initialize path $\Pi_{fw} = \text{NULL}$;
- (2) Initialize FWPerfectConverge = FALSE;
- (3) Initialize overRun = FALSE;
- (4) $q_{new} = q$;
- (5) Compute constants of steering function;
- (6) **while** (not FWcollide and not overRun and not converge)
- (7) $q_{new} = \text{AdvanceForerunner}(q_{new}, q_{fw})$;
- (8) FWcollide = **CollisionChecking** (q_{new}, W);
- (9) **if** (not FWcollide) **then**
- (10) overRun = overImage (q_{new}, q_{fw});
- (11) FWconverge = **ConvergeChecking** (q_{new}, q_{fw});
- (12) **if** (not overRun and converge) **then**
- (13) FWPerfectConverge = TRUE;
- (14) Append q_{new} to Π_{fw} ;
- (15) **end while**;

Figure 5.2: The Algorithm for Forward Forerunner Simulation Applied in End-Portion Motion Planning.

Algorithm ConvergeChecking (q, q_{ref})

Input: q : a configuration; q_{ref} : a reference path;

Output: TRUE / FALSE

- (1) $\Delta d = |\text{the closest distance from } q \text{ to path } q_{ref}|$
- (2) $\Delta \theta = |\theta - \theta_{ref}|$
- (3) $\Delta k = |k - k_{ref}|$
- (4) **if** ($\Delta d < \epsilon$ and $\Delta \theta < \epsilon$ and $\Delta k < \epsilon$) **then**
- (5) return (TRUE);
- (6) **else**
- (7) return (FALSE);

Figure 5.3: The Algorithm for Convergence Checking in Path Tracking.

2. Collision Detection

While the forerunner is running in end-portion motion planning, collision detection is being performed in each advancing step. This is rather an important step in motion planning because the safety of the planned motion relies on it completely. In this dissertation, we assume that the robot is an autonomous vehicle with rigid body. A convex hull [23] can be constructed to enclose the shape of the robot. In motion planning, we compute the robot's current configuration based on its center. Thus each vertex of the convex hull can be calculated by adding the offsets to the coordinates of the robot's center. As defined in Chapter V, the free space is the compliment of the union of obstacles in the world. Thus the robot's motion is considered collision free if and only if all vertices of the convex hull fall into the free space.

Therefore in local motion planning, instead of checking possible collision with arbitrary obstacles, the collision detection task can be carried out by checking the convex hull with decomposed-regions. If all vertices of convex hull are inside of the regions in each sampling step, the collision will never happen. Now it comes to the question: Is it necessary to require the robot's motion having its trajectory always fallen into the regions on the global path class? Ideally, the robot's trajectory is passing through region by region along the global path class. For instance, if the global path class contains the regions R_1, R_2, \dots, R_n , the robot should move from the initial region R_1 then R_2, \dots , and finally stops at the final configuration in the region R_n . The robot is not expected to run into a region which is not in the sequence of the global path class. However, there will be some exceptions at end-portion. Figure 5.4 illustrates an example of the robot's motion in the initial region. In the figure, the region R_i is the initial region and R_j is its next region in the global path class, but the region R_k is not any region in the path. The broken curve shown on the figure is a possible traveling trajectory from the initial configuration q_s to the next region. Although

part of this trajectory falls into the region R_k which is not the region in the path, the motion traveling along this trajectory is obviously a safe one and probably a desirable one. This example suggests that to plan a safe motion, it is not necessary to keep the trajectory always in the regions listed on the global path class.

Since the motion is continuous, if the sampling step is small enough, the robot's position change will be relatively small in either real time or simulation. Thus, we can assume that the next position is either in the current region or the region adjacent to the current one. The current region, to which the robot's current position belongs, and the regions adjacent to the current one, are considered related regions. We conclude that collision detection can be done by checking robot's position with the related regions in each sampling step. If robot's position is in one of those related regions, it is appropriate to declare that there is no collision in current step. Figure 5.5 shows the algorithm for the collision detection.

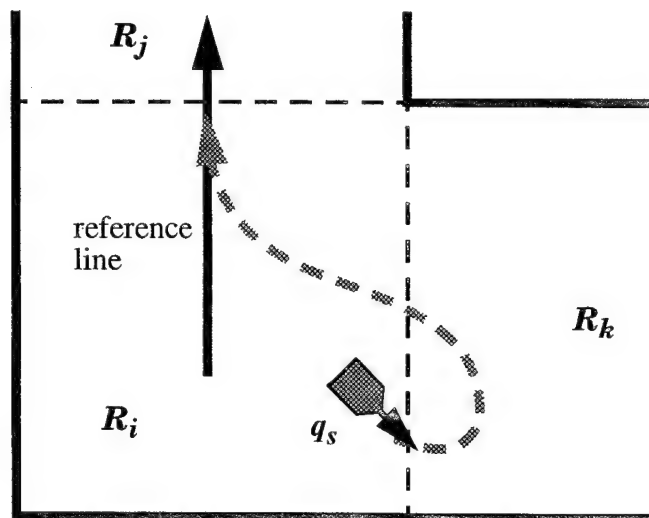


Figure 5.4: An Example of the Possible Motion Planning in the Initial Region.

Algorithm *collisionChecking* (q, W)

Input: q : a configuration; W : a world model;

Output: TRUE / FALSE

```
(1) Find the region  $R$  of current configuration  $q$ ;  
(2) if (no region found) then  
(3)   return (TRUE);  
(4) for all regions  $R_j$  adjacent to  $R$  do  
(5)   if  $q$  in region  $R_j$  then  
(6)     return (FALSE);  
(7) end for;  
(8) return (TRUE);
```

Figure 5.5: The Algorithm for Collision Detection.

3. Backing-up Motion Simulation

A forward motion is always preferable in motion planning. However, in some situation, forward motion may not be able to accomplish the task at the end-portion of entire motion planning. For instance, if all possible forward motion can not avoid collision with the obstacles, then other solutions will be needed. For a robot with nonholonomic constraints, the backing-up motion is the only solution. Figure 5.6 illustrates the situation in which the backing-up motion is required.

The Forerunner Simulation is applied in forward motion simulation as described in Subsection 1 of this section. The backing-up motion simulation can also be constructed by modifying Forerunner Simulation template in Figure 3.14. The purpose of backing-up motion simulation is to find the positions along the backing-up trajectory where a collision-free forward motion can start. For this purpose, in constructing backing-up motion simulation the followings are considered:

- Backing-up motion itself must be a collision free motion.

- In each sampling step, the forward forerunners should be issued to check whether it is possible to have a collision-free forward motion from the current configuration, while simulation is running backward. The Figure 5.7 illustrates the backing-up simulation with forward forerunner simulation.

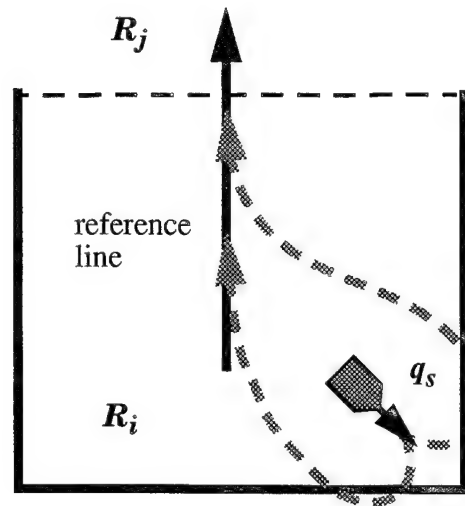


Figure 5.6: Unavoidable Collision with Forward Motion in the Initial Region.

While the simulation is running, both backward trajectory and forward trajectory are saved for planning use. Although the forward forerunner may complete a full simulation in each backing-up step, only the last forward trajectory will be kept.

The backing-up motion simulation terminates in one of following conditions:

- When backing-up motion collides with obstacles.
- When the virtual robot's orientation is identical to the orientation of its main (forward forerunner's) reference line.
- When the forward forerunner converges to its reference line before the point the reference line specified.

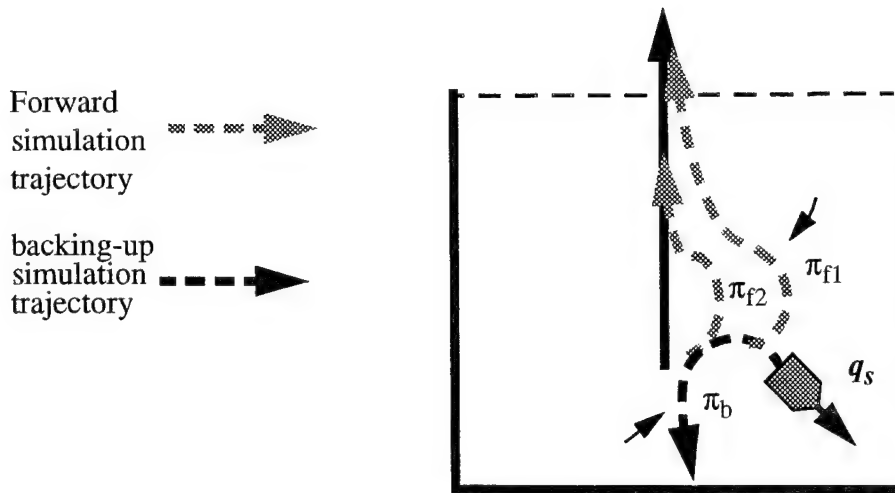


Figure 5.7: Backing-up Motion Simulation with Forward Forerunner Simulation.

If the backing-up simulation terminates with collision detected, it suggests that more forward or backward motions will be needed to complete the planning. We will discuss this in more detailed in Section B.

The second termination condition stops the backing-up simulation when the orientation of the virtual backing-up robot reaches the same orientation of forward reference line. Then the forward motion can be planned to track the line specified by the last configuration. Even though this motion may not cross the exit border at the originally planned crossing point, it will be an acceptable safe motion. This is because it crosses the border with an orthogonal orientation, so that the motion in the next region can be planned accordingly without any unexpected interference. Figure 5.8 illustrates this case.

The third termination condition will be met when the forward forerunner simulation makes its tracking trajectory converge to the main reference line before the point its configuration defines. This kind of convergence is called a *perfect convergence* because a forward motion then can be planned to track the reference line from the configuration the backing-up motion reaches. This forward motion will be able to cross the exit border at the planned crossing point. The trajectory π_{f2} in Figure 5.7 is an example of this case.

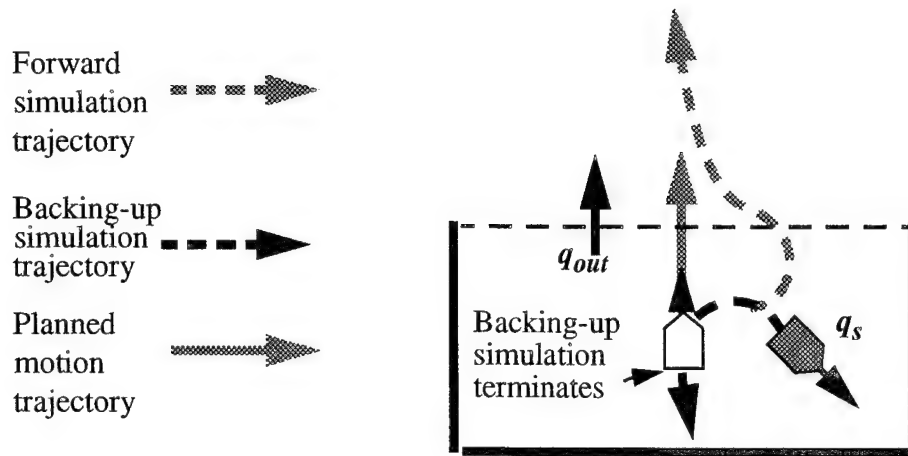


Figure 5.8: Backing-up Motion Simulation Terminates at the Configuration Where Its Orientation Is Identical to the Exit Orientation

Figure 5.9 and Figure 5.10 show the backing-up motion simulation algorithms. The algorithm for subroutine **BackUpForerunner** in Figure 5.10 is similar to the algorithm for **AdvanceForerunner** in Figure 3.15. The differences are (i). in line (1) of Figure 5.10, the curvature change will be computed by Eq 3.6 and Eq 3.7 based on planning a backward motion. Thus the virtual robot's orientation must be reversed. As a result, the orientation θ in Eq 3.7 will be replaced by $\theta + \pi$ in computing the curvature change, dk / ds . (2). In line (2), computing new curvature and orientation change by Eq 3.8 and Eq 3.9, the increment step Δs of Eq 3.8 and Eq 3.9 has to be a negative value to make the position change backward. The rest of computations are the same as described in Chapter III Section E.2. The subroutine **ForwardSimulation** in Figure 5.9 is presented in Figure 5.2.

C. INITIAL MOTION PLANNING

In Chapter III and IV, we describe mid-portion motion planning in a single region. This is obviously a workable solution, because the entire motion is linked by the motion passing through crossing point of borders shared by neighboring regions in the global path class. When we discuss the end-portion motion planning, we notice that it may involve not

only the initial or final region but also the regions next to initial or final one. This is because the initial or final configuration is arbitrary. Therefore, the robot moving from the initial region to the next one does not necessarily cross the first border at its center and with orthogonal orientation. This would be a conflict with essential idea of mid-portion motion planning. When this does happen, the motion planning in the region next to the initial region or final region will be considered a part of end-portion motion planning.

Algorithm BackUpSimulation ($q, q_{bw}, q_{fw}, \sigma, W$)

Input: q : an initial configuration; q_{bw} : a reference path for backing-up simulation;
 q_{fw} : a reference path for forward; σ : a smoothness; M : a world model;
Output: The set of data including the path of backward motion, the path of forward motion, and collision indicator for backward motion.

- (1) Initialize FWPerfectConverge = FALSE;
- (2) Initialize BWcollide = FALSE;
- (3) Initialize BWfinished = FALSE;
- (4) $q_{new} = q$;
- (5) Compute constants of steering function with smoothness σ_{min} ;
- (6) **while** (not BWcollide and not BWfinished and not FWPerfectConverge)
- (7) $q_{new} = \text{BackUpForerunner}(q_{new}, q_{bw})$;
- (8) BWcollide = CollisionChecking (q_{new}, W);
- (9) **if** (not BWcollide) **then**
- (10) **if** ($\theta_{new} \neq \theta_{fw}$) **then**
- (11) ForwardSimulation ($q_{new}, q_{fw}, \sigma, W$);
- (12) **else** BWfinished = TRUE;
- (13) Append q_{new} to path Π_{bw} ;
- (14) **end while**;

Figure 5.9: The Algorithm for Backing-up Motion Simulation in End-Portion Motion Planning.

Based on the line tracking of steering function, the Forerunner simulation described in Section A are applied to the initial motion planning. Let $q_s = (p_s, \theta_s, k_s)$ be the start configuration, and $q_{out} = (CP_{out}, \psi_{out}, k_{out})$ be the configuration defined by the crossing point, PC_{out} , and orientation, ψ_{out} , of the exit border in initial region. Let d_V denote the

closest distance between start configuration q_s and the reference line. (For example in Figure 5.11 reference line is the line defined by q_{out})

Algorithm *BackUpForerunner* (q, q_{ref})

Input: q : the current configuration; q_{ref} : the reference path;

Output: next configuration

- (1) Compute curvature change in backward direction;
- (2) Compute new curvature and orientation change;
- (3) Compute new configuration of next step;
- (4) return (new configuration)

Figure 5.10: The Algorithm for Backing-up Forerunner in One Step.

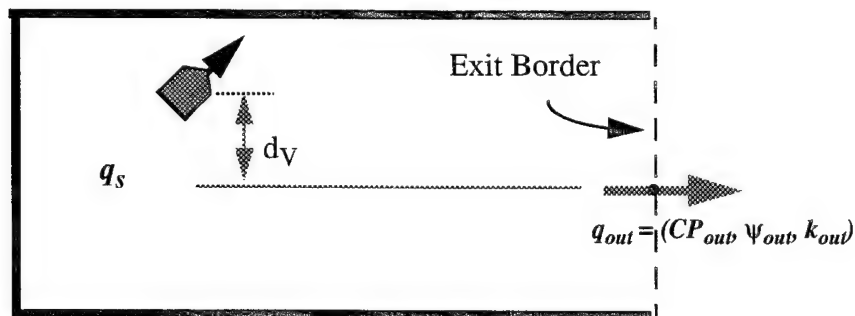


Figure 5.11: An Example of the Initial Region.

The initial motion planning will be performed by following basic steps:

- Determine a reference line q_{ref} .
- Determine smoothness σ .
- Run forward Forerunner Simulation.
- If no collision-free forward motion is possible, run backing-up motion simulation.

1. Determination of Reference Line

The reference line is used to drive the virtual robot to a desired direction in Forerunner Simulation. In the initial motion planning, the crossing position of the exit border, CP_{out} , is predetermined (in most cases, it is the center of the border). With the orientation of exit border, ψ_{out} , it defines the exit configuration $q_{out} = (CP_{out}, \psi_{out}, k_{out})$. It is desirable to plan a motion that leaves the region at exit configuration (at the position CP_{out} and with orientation ψ_{out}) because the exit configuration in this region is exactly the entrance configuration of next region on the path. Thus the line specified by the exit configuration of the initial region will be the first choice in determining reference line of forward Forerunner Simulation. In some cases, we may need to adopt a reference line other than the exit configuration of initial region. In this dissertation, four different lines will be considered useful in end-portion motion planning. We define these lines as follows:

- $q_1 = q_{out-init}$: The line specified by the exit configuration of initial region.
- q_2 : A line which passes through the position of initial configuration and is perpendicular to the $q_{out-init}$.
- $q_3 = q_{out-next}$: The line specified by the exit configuration of the region next to the initial region.
- $q_4 = q_{ref-next}$: The line computed as the reference line in the region next to the initial one by the method described in mid-portion motion planning.

The line q_1 and q_2 are illustrated in Figure 5.12(a). The figure shows the examples of two different start configurations, q_{s1} and q_{s2} , which need different reference line in motion planning. The Figure 5.12(b) illustrates the situation that reference line q_3 and q_4 are adopted. In the figure R_{init} is the initial region where the start configuration locates, and the region R_{next} is the region next to the initial region on the global path class.

In this dissertation a line is specified by a configurations defined as $q = (p, \theta, k)$, where p can be any point on the line it defined. However, for planning convenience, when

a line is adopted as the reference lines in line tracking, we would like to use a more specific configuration to specify this line. That is the point p that will be the farthest point on the line where we expect a line tracking to converge. The reference line candidates q_1 and q_3 are on the borders of the regions. The points on the specifying configuration are already the farthest points on the lines, because beyond those points the lines will be out of their regions. For line q_2 and q_4 , the expected converge points can be calculated. The Figure 5.13 illustrates how the expected converge point for line q_2 is calculated. The line q_2 is designed as a line perpendicular to the line specified by q_{out} . Its orientation can be computed as $\theta_2 = \psi_{out} \pm \pi/2$.

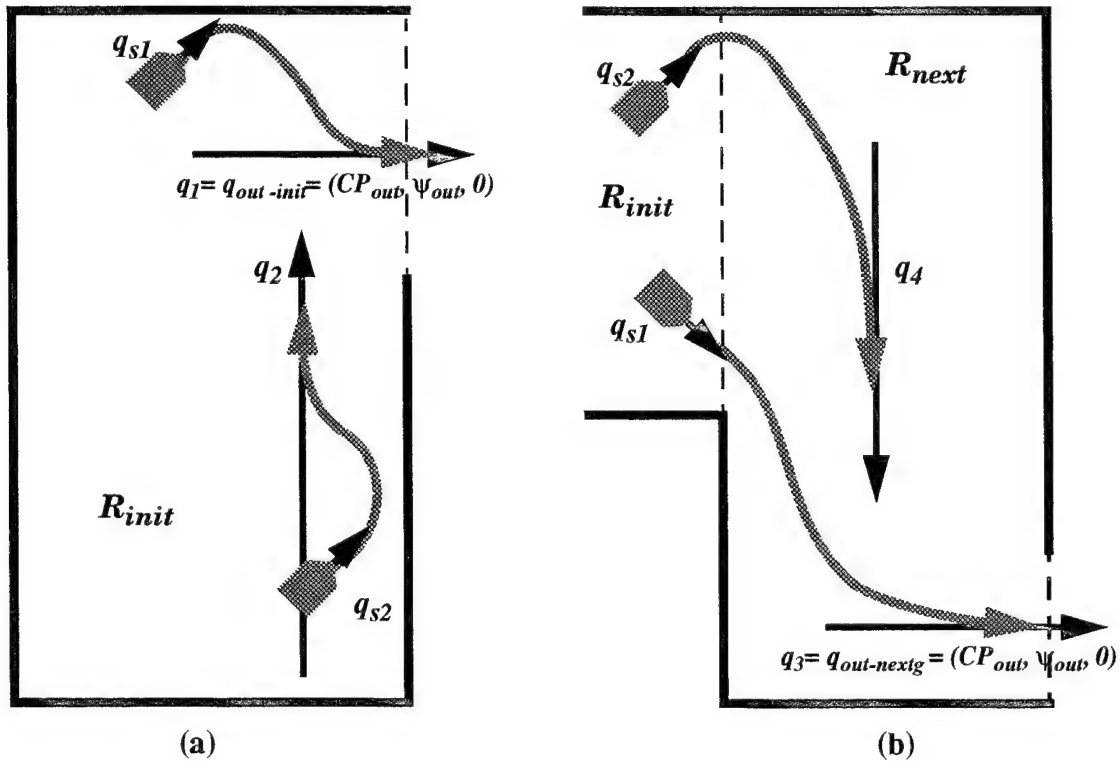


Figure 5.12: The Reference Lines in the Initial Motion Planning.

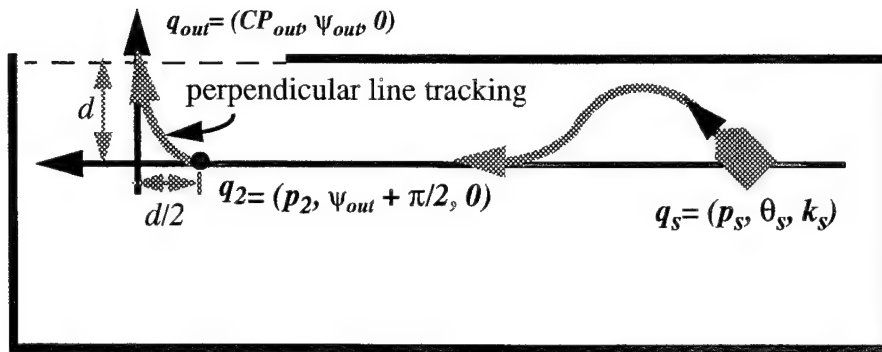


Figure 5.13: An Example of Reference Lines q_2 in the Initial Motion Planning.

For a smooth motion planning in the example of Figure 5.13, eventually there would be a perpendicular line tracking from a point p_2 on q_2 to line q_{out} as indicated in the figure. For perpendicular line tracking using minimum desirable smoothness, the distance to converge is twice of the distance from the starting position to the reference line as shown on Table 4.3 and Table A.2. Since the line q_2 is also designed to pass through the initial position, the distance to converge, d , can be calculated easily. As a result, the point p_2 is then computed easily. Therefore the line q_2 is defined as $(p_2, \psi_{out} + \pi/2, 0)$ in the example of Figure 5.13.

The reference lines described above are for the forward Forerunner Simulation. Once the reference line is determined, with proper smoothness the simulation starts a forward Forerunner Simulation. If the forward simulation can not complete the planning, then the backing-up motion simulation will be issued. As forward one, the backing-up motion simulation needs a reference line to start its work. Since the backing-up simulation is a supplemental tool whose main task is to make forward simulation possible to fulfill the end-portion motion planning, the forward reference line can be considered as its goal. So that while the backing-up simulation is running, the virtual robot's orientation is changing and gradually heading toward the direction of the forward reference line. This suggests that the reference line for backing-up motion simulation must have its orientation opposite to

the orientation of forward reference line. We conclude this subsection with defining the reference line for backing-up motion simulation as:

$$q_{backup} = (p_s, \theta_{forward} + \pi, 0) \quad (\text{Eq 5.1})$$

where p_s is the position of start configuration, and $\theta_{forward}$ is the orientation of predetermined forward reference line.

2. Dynamic Smoothness in Initial Motion Planning

In order to simply the initial motion planning among various situations with start configuration, the minimum desirable smoothness will be computed dynamically depending on the relationship between the position of start configuration and the reference line. As shown on Table A.1 and Table A.2 in Appendix A, the Max-Min smoothness for various orientation is

$$\sigma = 0.22 * d_V \quad (\text{Eq 5.2})$$

where the d_V is the closest distance from the start configuration to the reference line. The smoothness computed by Eq 5.2 may satisfy most of line tracking cases. However, the distance d_V can be very small, even close or equal to zero, but the smoothness used in steering function can not be too small. Therefore a lower bound of minimum smoothness is required. According to our experience in using steering function, the smallest smoothness that makes line tracking work is twice as much as the distance the robot advance in a motion control cycle (a step in simulation). For instance if each step is 0.1 cm, then the minimum smoothness is $\sigma = 0.2$. As we know, the smaller the smoothness is, the sharper its turning will be. To avoid unreasonable sharpness, we set the overall minimum smoothness to be

$$\sigma_{min} = 5.0 \quad (\text{Eq 5.3})$$

Therefore, the smoothness is determined as Eq 5.4

$$\sigma = \max(\sigma_{min}, 0.22 * d_V) \quad (\text{Eq 5.4})$$

3. Motion Planning Simulation

Since four lines can be taken as a reference line for forward forerunner simulation in initial motion planning, we will describe the motion planning simulation with those different reference lines individually in this section. In addition, the backing-up motion planning is also discussed in detailed in this section. We name the motion planning simulations as follows:

- The First Planning Simulation.
- The Second Planning Simulation.
- The Third Planning Simulation.
- The Fourth Planning Simulation.
- The Backing-up Motion Planning Simulation.

The initial motion planning begins with First planning simulation, then the other planning simulations may follow if necessary.

a. The First Planning Simulation

This forward forerunner simulation takes the line specified by the exit configuration, q_1 , of initial region as its reference line. In the forward forerunner simulation, the alternative tracking methods in Two-way Line Tracking (Appendix A) will be tried after the normal tracking fails to find a motion with its trajectory perfectly converging to the reference line. If a trajectory with perfect convergence and no collision is found, the First planning simulation is done. Then the motion will be planned to track the line q_1 directly. Figure 5.14 illustrates this planning.

If there is a collision in the First planning simulation, we may need to try other simulations which is the Second Planning Simulation. We will discuss this simulation later in this section.

If this simulation has no collision detected, but its trajectory does not converge in the region. This suggests that the start configuration is close to the exit border.

It may be better to track a reference line in the region next to the initial one. Therefore the Third Planning Simulation follows this case.

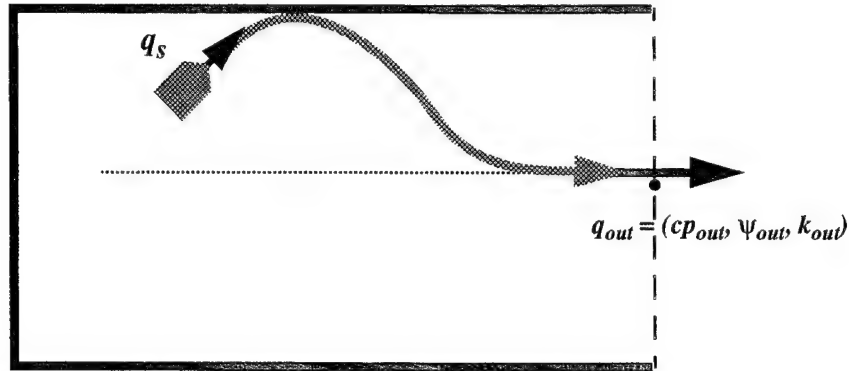


Figure 5.14: An Example of Forward Forerunner Simulation with No Collision and Converging in the Initial Region.

b. The Second Planning Simulation

Figure 5.15 shows the situations when a collision will be detected in forward forerunner simulation. In the figure, q_{s1} and q_{s2} represent start configuration in two different case. In the case with start configuration q_{s1} , the image of start configuration on the exit edge falls on the border. When there is a collision in forward simulation, there is no other way which leads to the exit border but making a backward motion first. Thus in this case, the backing-up motion planning simulation will be tried. It will be discussed in the subsection e. In the case with start configuration q_{s2} , the image of start configuration on the exit edge is not on the border. Although tracking the line specified by the exit configuration will result a collision in forward simulation, there might be a solution other than backward motion. This solution is to compute an alternative reference line q_2 as described in Section C.1, and then try a simulation of tracking the alternative reference line. If the simulation makes its tracking trajectory a perfect convergence, the initial motion can

be planned to track q_2 first followed by tracking the line of exit configuration as Figure 5.16.

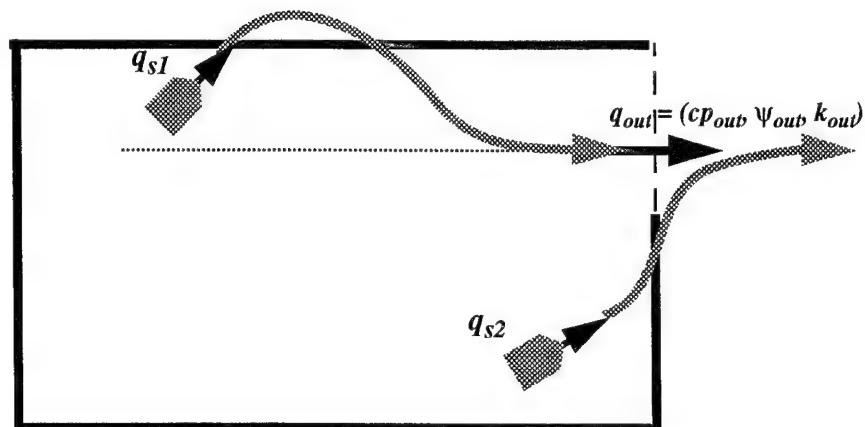


Figure 5.15: The Cases of Forward Forerunner Simulation with Collision Detected

If the simulation with alternative reference line can not make a perfect convergence trajectory, the backing-up motion planning simulation then follows to seek the solution.

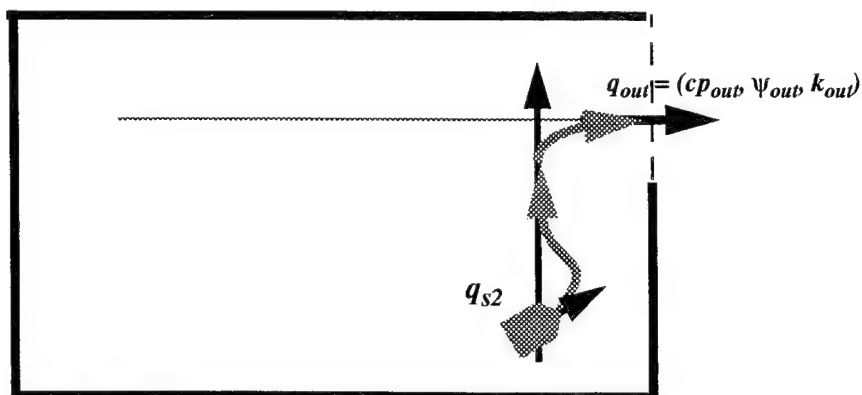


Figure 5.16: An Example of Alternative Reference Line in Solving Forward Forerunner Simulation with Collision Problem.

c. The Third Planning Simulation

This simulation is conducted after knowing the start configuration is too close to the exit border to make tracking the reference line converge. It takes the exit configuration q_3 in the next region on the global path class as its reference line for the first trial. If this simulation find that the tracking trajectory is non-collision and perfectly converges to the reference line, the simulation terminates and the motion is planned to track the reference line q_3 as illustrated in Figure 5.17.

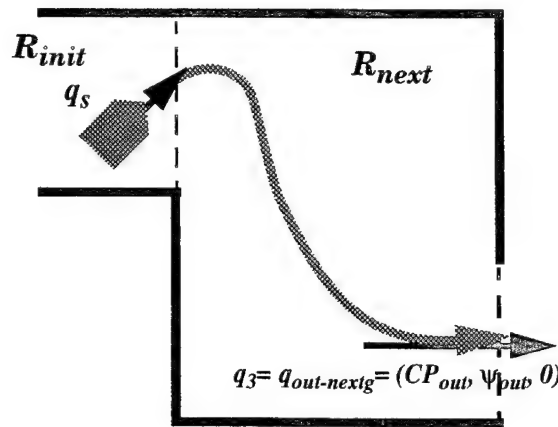


Figure 5.17: The Example of Initial Motion Planning with Reference Lines in the Region Next to the Initial Region.

If the Third Planning Simulation find that it is not possible to have a perfect convergence in the tracking trajectory with q_3 as its reference line, the other alternative reference line in the second region of the global path class will be computed and the Fourth Planning Simulation will follow to try to find the solution.

d. The Fourth Planning Simulation

The line q_4 will be computed and used as the reference line in this simulation. This line is defined in the region next to the initial one. Consulting the planning rule in mid-portion motion planning, we can identify the reference line in that region in mid-portion motion planning. It can be one of the center lines or the line specified by

entrance configuration or exit configuration. Since the entrance and exit border of the region are known, the position p_4 on the line $q_4 = (p_4, \theta_4, 0)$, where we expect the line tracking to converge, can be computed as described in Section C.1.

The Fourth Planning Simulation is a forward forerunner simulation. It tracks the reference line q_4 from start configuration q_s . If it outputs a trajectory of perfect convergence, the initial region motion will be planned to track the reference line q_4 until the robot reaches p_4 . Then a perpendicular line tracking from q_4 to the line of exit configuration follows. Figure 5.18 illustrates this planning. If no such a trajectory is found, it goes to performing backing-up motion planning simulation.

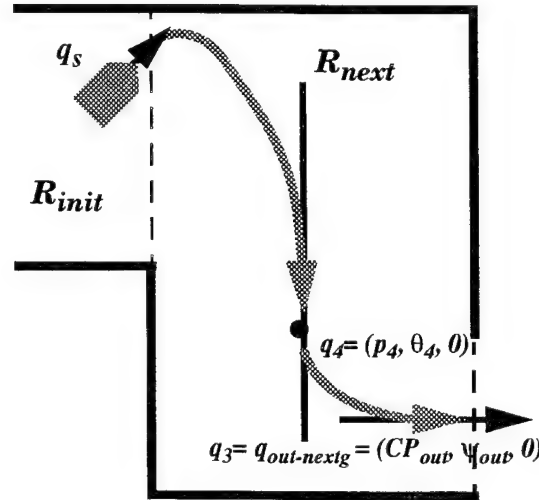


Figure 5.18: The Example of Initial Motion Planning with Reference Lines in the region Next to the Initial Region.

e. The Backing-up Motion Planning Simulation

When a single forward motion is not possible to complete the initial motion planning, the backing-up motion simulation will be the only solution to this problem. The goal this simulation is either the virtual robot backs up to the orientation of the forward reference line or the forward simulation achieves a perfect convergence. The algorithms for backing-up motion simulation is presented in Section C of this chapter. This simulation

requires two predetermined reference lines. One is for backward motion and another one is for forward motion. Basically the reference line for forward motion is the line defined by exit configuration in initial region for most cases. However, it can be a line which is perpendicular to the line of exit border as referred q_2 in Section C. The simple rule for determining forward simulation reference line is: if the image of start configuration on the exit edge is not on the border, then compute a line q_2 for the reference line. Otherwise, the reference line will be the line specified by the exit border. For backing-up motion simulation, the reference line is as Eq 5.1 in Section C.1.

The smoothness for forward forerunner simulation is computed as Eq 5.4. For backing-up motion simulation, the minimum smoothness as Eq 5.3 will be applied to make the backward motion achieve its goal as quickly as possible.

After the reference lines and smoothness are determined, the backing-up simulation algorithm with forward simulations inside of it can perform its first simulation. Once the goal is reached, the simulation stops. Let q_s , q_{bw} , q_{ref} be the start configuration, the configuration the backing-up simulation stops and forward reference line respectively. The initial motion is thereby planned as follows: (i) backing-up from q_s to q_{bw} . (ii) perform a forward motion tracking q_{ref} . Figure 5.19 demonstrates an example of initial motion planning with single backing-up simulation. In the example, the backing-up simulation terminates at the configuration where the forward forerunner can make a perfect convergence in tracking the forward reference line.

In the case that the initial motion planning can not be completed by single backing-up simulation, a forward simulation and a backing-up simulation will be repeated in order until the goal is reached. In the repeated back and forth simulation, a few works will be processed before next simulation begins:

- The last configuration, where the backing-up or forward simulation stops, is taken as an initial configuration for next simulation and its curvature is reset to zero. This is necessary and possible because back and forth motion is a

combination of many discrete motions. The robot's curvature can be discontinuous.

- The smoothness for the forward simulation will be recomputed according to its distance to the reference line.
- The reference line for the backing-up simulation will be recomputed according to the new start configuration:

There is a possibility that neither backing-up nor forward simulation can be conducted. In this case, the simulation will terminate and report an error.

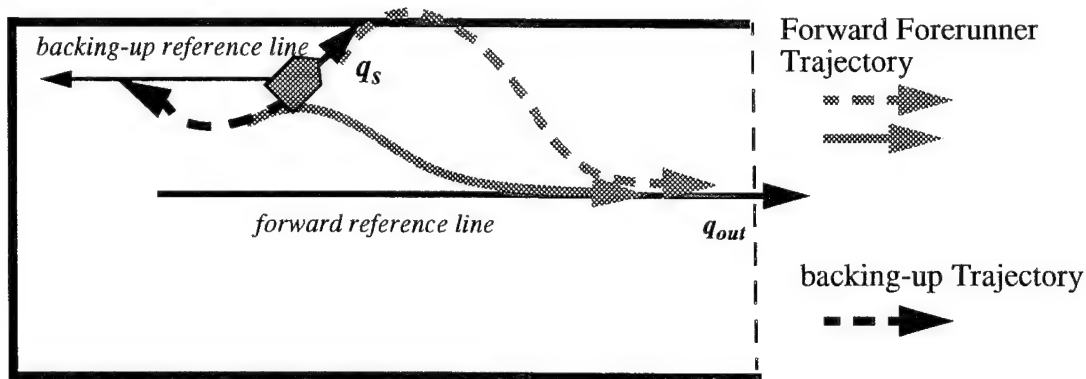


Figure 5.19: An Example of Backing-up Motion Simulation in the Initial Motion Planning.

4. Initial Motion Planning Algorithms

Some notations will be used in the initial motion planning algorithms. They are listed as follows:

- q_s : Start configuration;
- $q_{ref} = (p_{ref}, \theta_{ref}, 0)$: current reference line;
- $q_{ref1} = (p_{ref1}, \theta_{ref1}, 0)$: the reference line for backing-up simulation;
- $q_{ref2} = (p_{ref2}, \theta_{ref2}, 0)$: the reference line for forward forerunner;
- q_1, q_2, q_3, q_4 : the lines defined in Section B.1;

- d_V : the distance from current configuration to the reference line;
- σ : current smoothness;
- σ_{min} : The minimum smoothness as defined in Eq 5.3;
- Π_{fw} : a path of a sequence of configurations used to store a forward trajectory;
- Π_{bw} : a path of a sequence of configurations used to store a backing-up trajectory;
- backup: a flag indicating whether backing-up motion is needed;
- FWcollide: a flag indicating whether a collision exists in forward simulation;
- FWPerfectConverge: a flag indicating whether forward motion converges perfectly;
- BWcollide: a flag indicating whether a collision exists in backing-up simulation;
- BWpath[]: a structure used to store backing-up paths;
- FWpath[]: a structure used to store forward motion paths.

The initial motion planning algorithms are presented as the followings:

Algorithm InitialMP (q_s , path, W)

Input: q_s : initial configuration; path: a global path class; W: a world model

Output: Motion planning data structure

- (1) backup = FALSE;
- (2) **FirstPlanning** (q_s , path, W)
- (3) **if** (backup) **then** /* back-up motion is needed */
- (4) **BackUpPlanning** (q_s , path, W);

Algorithm FirstPlanning (q_s , path, W)

Input: q_s : initial configuration; path: a global path class; M: a world model

Output: Motion planning data structure

- (1) $q_{ref} = q_1$; /* the line specified by exit configuration in initial region */
- (2) compute d_V ;
- (3) $\sigma = \max(\sigma_{min}, 0.22 * d_V)$;
- (4) **ForwardSimulation** (q_s, q_{ref}, σ, W);
- (5) **if** (not FWcollide) **then** /* no collision */
- (6) **if** (FWPerfectonverge) **then** /* converge perfectly*/
- (7) Motion Planning: Tracking q_{ref} from q_s with smoothness σ .
- (8) **else** /* not converge before p_1 on line q_1 */
- (9) **ThirdPlanning** ($q_s, path, W$);
- (10) **else** /* collision in initial region */
- (11) **SecondPlanning** ($q_s, path, W$);

Algorithm **SecondPlanning** ($q_s, path, W$)

Input: q_s : initial configuration; path: a global path class; M : a world model

Output: Motion planning data structure and backup: a boolean variable

- (1) **if** (image of start configuration q_s is on exit border) **then**
- (2) backup = TRUE;
- (3) **else**
- (4) $q_{ref} = \text{compute forward reference line } q_2$;
- (5) $\sigma = \sigma_{min}$;
- (6) **ForwardSimulation** (q_s, q_{ref}, σ, W);
- (7) **if** ((not FWcollide) and (FWPerfectConverge)) **then**
- (8) Planning:
- (9) 1. Tracking q_{ref} from q_s with smoothness σ until reaches q_{ref} .
- (10) 2. Tracking q_1 from q_{ref} .
- (11) **else**
- (12) backup = TRUE;

Algorithm **ThirdPlanning** ($q_s, path, W$)

Input: q_s : initial configuration; path: a global path class; W : a world model

Output: Motion planning data structure and backup: a boolean variable

- (1) $q_{ref} = q_3$; /* q_{out} of the region next to initial region */
- (2) compute d_V ;
- (3) $\sigma = \max(\sigma_{min}, 0.22 * d_V)$;
- (4) **ForwardSimulation** (q_s, q_{ref}, σ, W);

- (6) **if** ((not FWcollide) and (FWPerfectConverge)) **then**
- (7) *Planning: Tracking q_{ref} from q_s with smoothness σ ;*
- (8) **else**
- (9) **FourthPlanning** (q_s , path, W);

Algorithm **FourthPlanning** (q_s , path, W)

Input: q_s : initial configuration; path: a global path class; W: a world model

Output: Motion planning data structure and backup: a boolean variable

- (1) $q_{ref} = \text{compute } q_4$; /* reference line of next region */
- (2) compute d_V ;
- (3) $\sigma = \max(\sigma_{min}, 0.22 * d_V)$;
- (4) **ForwardSimulation** (q_s , q_{ref} , σ , W);
- (6) **if** ((not FWcollide) and (FWPerfectConverge)) **then**
- (7) *Planning:*
- (8) 1. Tracking q_{ref} from q_s with smoothness σ until reaches p_{ref} ;
- (9) 2. Tracking q_3 from q_{ref} ;
- (10) **else**
- (11) backup = TRUE;

Algorithm **BackUpPlanning** (q_s , path, W)

Input: q_s : initial configuration; path: a global path class; W: a world model

Output: Motion planning data structure

- (1) **if** (image of q_s is on exit border) **then**
- (2) $q_{ref2} = q_1$;
- (3) **else**
- (4) $q_{ref2} = \text{compute forward reference line } q_2$;
- (5) compute d_V for forward forerunner with reference line q_{ref2} ;
- (6) $\sigma = \max(\sigma_{min}, 0.22 * d_V)$;
- (7) $q_{ref1} = (p_s, \theta_{ref2} + \pi, 0)$;
- (8) **BackUpSimulation** (q_s , q_{ref1} , q_{ref2} , σ , W);
- (9) counter = 1;
- (10) BWpath[counter] = store Π_{bw} ;
- (11) **if** (FWPerfectConverge) **then**
- (12) FWpath[counter] = store Π_{fw} ;
- (13) **while** (BWcollide and not done)

```

(14)  $q_{new} = \text{last configuration in } \Pi_{bw};$ 
(15)  $q_{new} = (p_{new}, \theta_{new}, 0);$ 
(16) ForwardSimulation ( $q_{new}, q_{ref2}, \sigma_{min}, W$ );
(17) if ( $q_{new} = \text{last configuration in } \Pi_{fw}$ ) then exit and report error;
(18) if (FWPerfectConverge) then
(19)    $FWpath[counter] = \text{store } \Pi_{fw};$ 
(20)    $done = TRUE;$ 
(21) else
(22)    $q_{new} = \text{last configuration in } \Pi_{fw};$ 
(23)    $q_{new} = (p_{new}, \theta_{new}, 0);$ 
(24)    $\sigma = \max(\sigma_{min}, 0.22 * d_V);$ 
(25)    $q_{ref1} = (p_{new}, \theta_{ref2} + \pi, 0);$ 
(26)   BackUpSimulation ( $q_{new}, q_{ref1}, q_{ref2}, \sigma, W$ );
(27)   increment counter;
(28)    $BWpath[counter] = \text{store } \Pi_{bw};$ 
(29)   if (FWPerfectConverge) then
(30)      $FWpath[counter] = \text{store } \Pi_{fw};$ 
(31) end while;
(32) Planning:
(33)   while ( $counter > 0$ )
(34)     backup tracking  $BWpath[counter];$ 
(35)     forward tracking  $FWpath[counter];$ 
(36)     decrement backupPath;
(37)   Tracking  $q_1$ ;

```

D. FINAL MOTION PLANNING

Let the path $\Pi = (R_1, B_1, \dots, R_n)$ be the global path class obtained in global path planning execution after a mission is given, where $n \geq 1$. The local motion planning for the final region R_n and probably the region near the final one is called **final motion planning**. The characteristics of the robot's motion in the final region of the path is that the robot must stop at the goal configuration specified in the given mission. A normal path tracking motion maybe suitable for some cases in which the orientation of goal configuration has small difference compared to the orientation of entrance border of the final region. In many other cases, however, the difference between those two orientations may be large. Then the simple path tracking motion may not be able to fulfill the task in the final region. One of

the methods for solving the final motion planning problem is proposed as *Bidirectional Motion Planning* [7].

As we notice that the *Bidirectional Motion Planning* is used to generate a reference path between two configurations. In the case of final motion planning, one of these two configurations is the goal configuration which is fixed and given by the mission. But another configuration is unknown until we determine where the Bidirectional Motion Planning commences will be better. In order to make a better or possible final motion planning using the Bidirectional Motion Planning algorithm, the determination of the starting configuration is important. However, this still remains an open question.

Another important task in the final motion planning is to plan a collision-free motion. The Bidirectional Motion Planning is an excellent approach in planning a path to connect two configurations. However, it is provided that the planning will be performed in an obstacle-free space. As we know, final motion planning is based on a global path class which relies on decomposed regions. That means the obstacles exist not only in the real world, but also in the world model we are working on. For those reasons, the Bidirectional Motion Planning is considered not practical in this dissertation.

Local motion planning is to plan a safe and symmetric path for robot to follow. The final motion planning is actually the mirror of the initial motion planning in the sense of symmetry. Therefore, we propose that the final motion planning be performed by the approach used in initial motion planning. Only by doing this, can the entire path planned from start configuration to goal configuration be symmetric to the path planned reversely.

To apply the motion planning approach used in initial planning to final motion planning, the algorithm described in Section C can be adopted with a minor modification. First of all, the goal configuration $q_s = (p_s, \theta_s, 0)$ needs to be replaced as Eq 5.5:

$$q_g = (p_g, \theta_g + \pi, 0) \quad (\text{Eq 5.5})$$

That is to reverse the orientation of the goal configuration. The same operation will be applied to the orientations which define the entrance configurations, q_{in} , of the final region

and the region prior to it. After then the entrance configurations will be rephrased as exit configurations. So as to the entrance borders and entrance edges. With all those orientations and names reversed, the final motion planning can be done by the algorithm presented below. The algorithms of all other subroutines called by the algorithm FinalMP will be similar to the algorithms described in Section C.

Algorithm FinalMP ($q_g, path, W$)

Input: q_g : goal configuration; $path$: a global path class; W : a world model

Output: Motion planning data structure

- (1) $backup = FALSE$;
- (2) $q_g = reversed\ q_g$;
- (3) $q_{out-init} = reversed\ q_{in}\ for\ final\ region$;
- (4) $q_{out-next} = reversed\ q_{in}\ for\ the\ region\ prior\ to\ the\ final\ region\ on\ the\ global\ path$;
- (5) **FirstPlanning** ($q_g, path, W$)
- (6) **if** ($backup$) **then** /* back-up motion is needed */
- (7) **BackUpPlanning** ($q_g, path, W$);

VI. YAMABICO HARDWARE ARCHITECTURE

This chapter introduces the hardware structure of the robot -- Yamabico-11 which is used to test most of our algorithms experimentally.

A. OVERVIEW

Yamabico-11 is a wheeled untethered indoor mobile robot for AI and robotics research. It has been developed at the Naval Postgraduate School over the last several years. However, the vehicle is a result of Dr. Yutaka Kanayama's long history of autonomous robotics research at the University of Electro-Communications, the University of Tsukuba, Stanford University, and the University of California at Santa Barbara [24], [25]. Its main CPU board consists of the SPARC microprocessor and a 16 Mbyte RAM storage and is mounted on a VME bus. Besides that, a dual-axis controller for two motors and two shaft encoders, a tailor-made sonar board, and a serial communication board are also mounted on the VME bus. One lap-top computer is used for a real-time input/output device. The size is 60(W) by 60(L) by 70(H) centimeters. It weighs about 60 kilograms. The differential drive kinematic architecture is used for the wheel system. Two 35 watts DC motors with shaft encoders are used with 1/24 gear boxes. Twelve 40KHz sonars and one CCD camera are mounted on board. Its power source is two 12 volt car batteries. When object code is downloaded from a UNIX system, the vehicle operates as an untethered (self-contained) autonomous robot. [19]. Figure 6.1 illustrates the Yamabico-11 hardware architecture.

B. IV-SPARC-33 CPU

The Ironics IV-SPARC-33 is a single processor, VMEbus Interface, CPU board. It contains a 25Mhz SPARC Integer Unit, a Floating Point Unit, and a Cache Controller and Memory Management Unit. The card installed in Yamabico has 64 Kbytes of cache, and 16 Mbytes of 80ns DRAM. It provides two RS-232 serial I/O ports, two programmable timers, and seven user-definable LEDs [26].

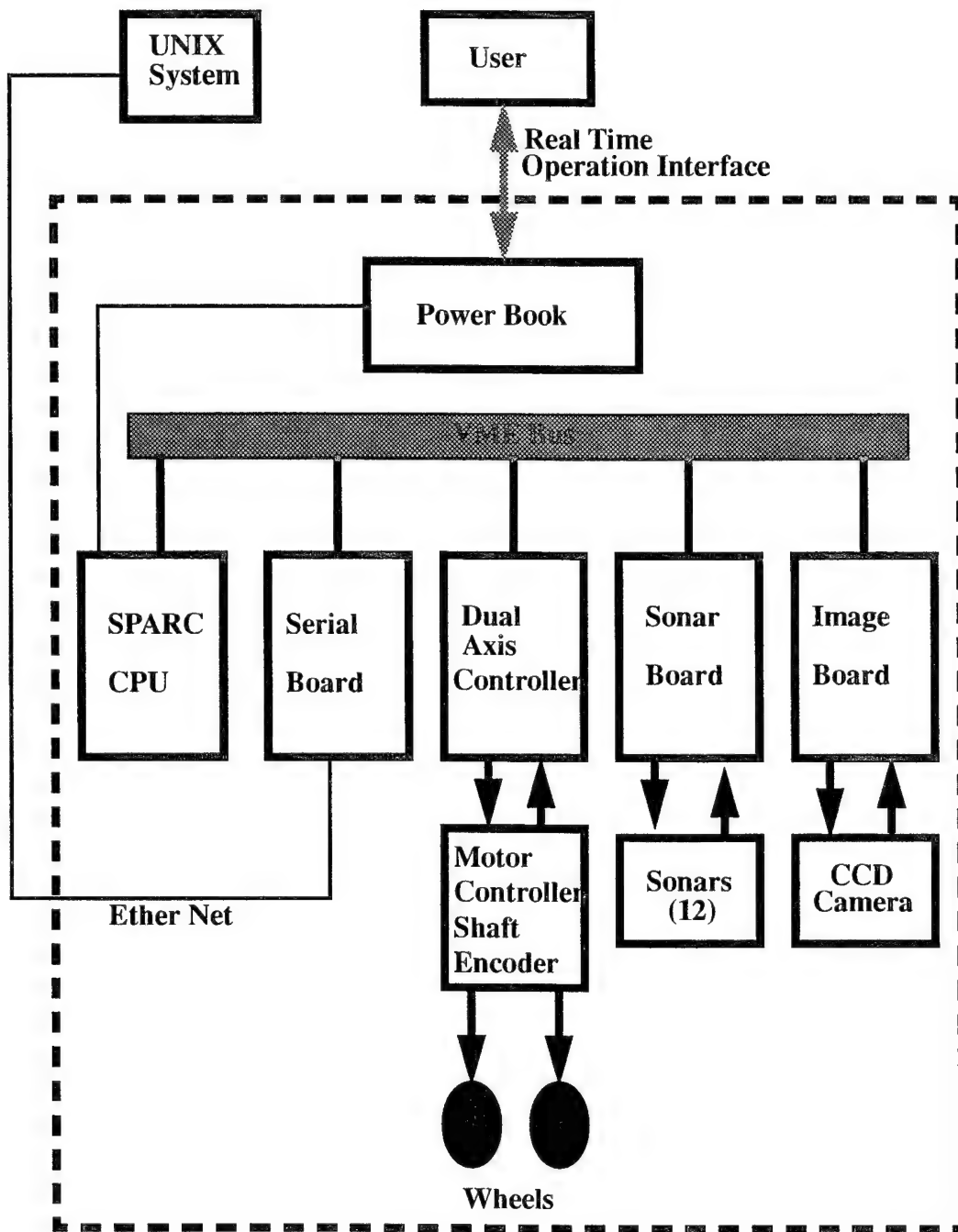


Figure 6.1: Block Diagram of Yamabico-II Hardware Architecture

1. Address Map

The Ironics SPARC board contains 16 Mbytes of physical memory, yet provides 32 bit addresses (4 GBytes). This 4 GByte address space is logically divided into several regions. The three most important regions are the Local DRAM, Region 3, and Local I/O (Figure 6.1).

a. Local DRAM

The Local DRAM is the physical memory present on the board, and is addressed from 0x00000000 to 0x00FFFFFF. The DRAM array is organized into two-word (64-bit) interleaved banks of 8 Mbytes each. Bidirectional latching data buffers direct to and from the two banks. The DRAM contains the SPARC trap table, ISPARCmon stack, uninitialized data segment (bss), Global Work Area, and all the target program segments. The target program (application program) is loaded by default starting at location 0x00018000. From this lower limit up, the text segment is first, followed by the initialized data and uninitialized data (bss). The region for target program stack starts from the address (DRAM TOP - 0x000000FF) because the top 256 bytes are reserved for mailbox interrupts. This stack is extended downward from the top of this region.

b. Region 3

Region 3 starts at the end of Region 2, and extends to the bottom of the EPROM space. The default configuration provides addresses from 0xFC000000 to 0xFF0000, however, only the upper boundary is fixed. The lower boundary may be changed by writing to the appropriate register as defined in the Ironics manual. MML11 currently does not change the default address map, but does provide for Region 3 to be VMEbus A16 addressable. All devices on the VMEbus are addressed from Region 3. Addresses are obtained by adding the 16-bit base address of a specific hardware device to the Region 3 offset of 0xFC000000. This includes the shaft encoders, quad serial boards, and sonar board.

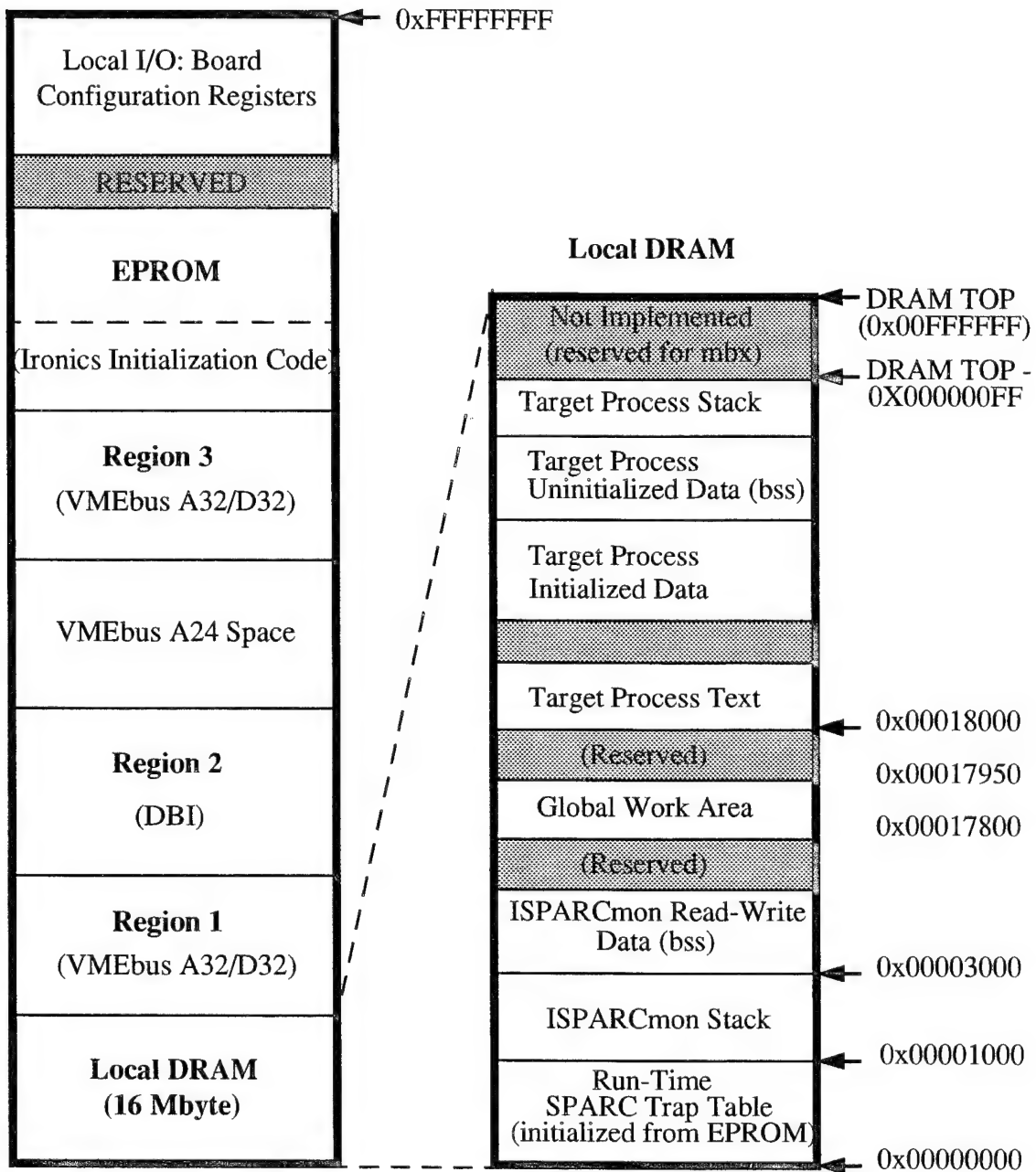


Figure 6.2: Memory Organization of Ironics IV-SPARC-33 board on Yamabico-11

c. Local I/O

The Local I/O region contains the addresses for the registers which control operation of the Ironics SPARC board. They are addressed from 0xFFFF0000 to 0xFFFFFFFF. Both limits are fixed.

2. Registers

There are two sets of registers which control the CPU board; one set consists of 8-bit registers, and the other consists of 16-bit registers. The 8-bit registers start at address 0xFFC00000, and the 16-bit registers start at address 0xFFD00000. The addresses, functions, and default settings are clearly defined in the Ironics users manual.

The value of these registers may be changed by writing directly to the appropriate address, or by using one of the two library functions provided with the CPU board. Similarly, their values can be read directly or through library functions. The current version of MML11 uses the direct write/read method in order to avoid additional overhead associated with a function call.

3. Interrupt Map

The Ironics SPARC CPU board emulates a 680x0-style interrupt-acknowledge cycle. A library function, `mk_handler()`, assigns a user defined interrupt handler to an appropriate interrupt vector. All interrupt vectors are defined in the Ironics users manual.

The syntax for this function is: `mk_handler (vector_number, interrupt_handler)`

4. Internal Interrupts

Internal Interrupts are those generated on the CPU board. The two most important are the Timer 1 and Timer 2 interrupts. Timer 1 can be set to provide interrupts at 50, 100, or 1000 hz. Currently, MML11 uses Timer 1 to provide the 10ms (100 Hz) motion control interrupt. Timer 2 provides a broader range of interrupts, and is currently unused. The interrupt vectors for Timer 1 and 2 are defined by the Local Interrupt Vector Base Register (0xFFFC0057).

5. External Interrupts

External Interrupts are those generated off the CPU board. The most important are from the Quad Serial Boards, and the Sonar Board, which are handled through the 7 VMEbus Interrupt Request lines. Currently the VMEbus cards are hardware jumpered to the following interrupts:

IRQ1: Serial Board 0, ports 1A and 1B (which are unused), and the #5 timer

IRQ2: Serial Board 1, ports 1A, 1B, 2A, 2B, which are all unused

IRQ3: Serial Board 0, ports 2A and 2B (host and old console connections)

IRQ4: Serial Board 1, #5 timer

IRQ5: Unused

IRQ6: No connection

IRQ7: Abort button

Since the Serial Board ports jumpered to IRQ2 are unused, this request line is being used to handle interrupts generated by the Sonar Board. The number of the IRQ does not reflect its priority. Priorities are set by modifying the appropriate IRQ Interrupt Control Register (0xFFFFC0007-0xFFFFC001F). Additionally, the VMEbus cards must be set to assert an interrupt vector as defined in registers 0xFFFFC0087-0xFFFFC009F.

C. SONARS

The sonar system mounted on Yamabico-11 consists of a sonar ranging board and a sonar array consisting of twelve Nippon Ceramic T40-16/R40-16 ultrasonic range finder emitter/receiver pairs arranged around the robot's perimeter. The ranging board is an Ommibyte OB68K VME I/O board that is controlled by an 8748 microcontroller. The sonar board is a separate input/output controller that makes the overall sensor process more efficient [21].

Currently, Yamabico's 12 sonars are positioned around the periphery of the robot with 30 degree increments from the central one on the robot's front edge. The actual positions of sonars are measured from the center of the robot with an imaginary Cartesian

Coordinate system. This positioning provides the most comprehensive sonar coverage.

Figure 6.3 illustrates the sonars' configurations.

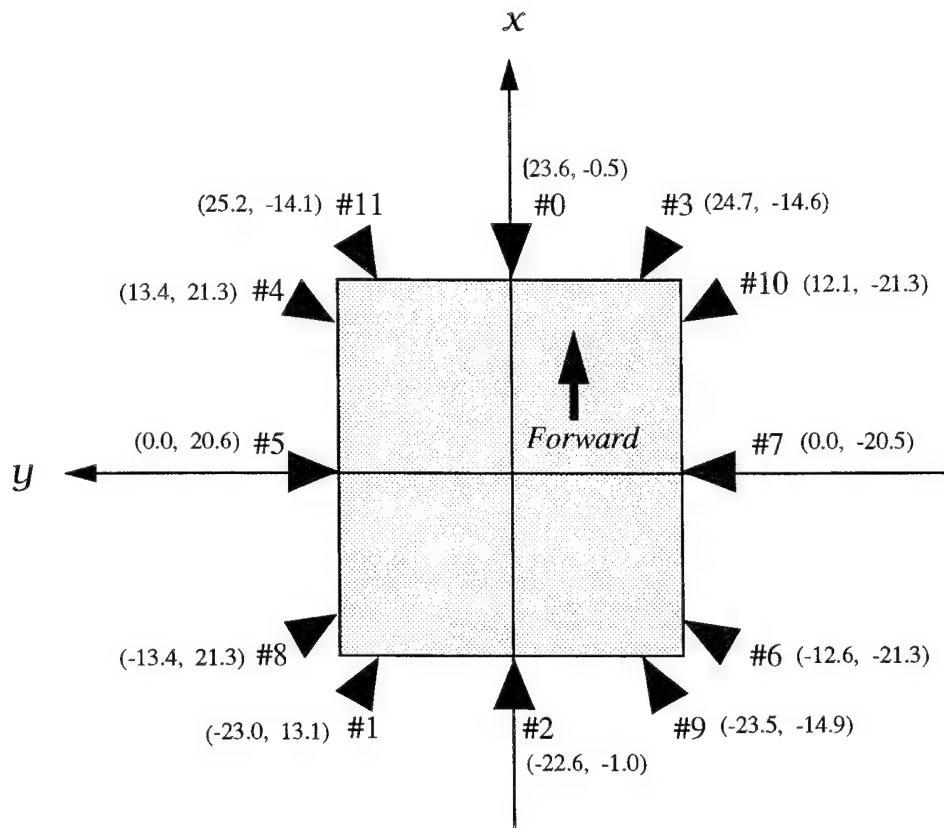


Figure 6.3: Yamabico-11 Sonar Configurations

1. Sonar Control

The sonar control board is actually a daughtercard which rides on a VME bus mothercard. The mothercard carries address decoders, bus drivers and interrupt control circuitry in the Bus Interface Module (BIM).

When the sonar has completed a ranging cycle an interrupt request is provided to the BIM. The BIM's control register holds information which determines whether an interrupt is to be generated or not, and if so which interrupt level is to be generated. Presuming an interrupt is generated, when the correct acknowledgment returns on the

address lines the BIM's vector register provides the vector table entry where the central processor may find the vector to the interrupt handler. The correct interrupt level, the interrupt enable bit and interrupt vector are loaded to the BIM during software initialization.

2. Sonar Grouping

In order to reduce sampling time the sonars are operated in logical groups of four. The sonars of a logical group are all pulsed simultaneously and thus the sampling time is reduced by a factor of four as compared to individual firing of the sonars. The sonars of each logical group are oriented in such a way as to:

- prevent mutual interference
- provide a "look" in all four directions from each group
- present a similar aspect from each sonar during a rotational scan

Thus, logical group 0 consists of sonars 0, 2, 5 and 7, group 1 consists of sonars 1, 3, 4 and 6; group 2 consists of sonars 8, 9, 10 and 11; and group 3 is a "virtual" group which consists of four permanent test values. The sonars of a group are symmetric about the robot's axis of rotation.

In addition to being logically grouped, the sonars are also physically grouped. The physical grouping of the sonars is made to distribute the electrical load over the driver boards evenly and thus minimize any electrical transients associated with operation of the sonar. The physical grouping connects sonars 0, 2, 8 and 11 to driver/amplifier board 1; sonars 4, 5, 6 and 7 to board 2; and sonars 1, 3, 9 and 10 to board 3. The reader will note that pairs of sonars from logical groups are assigned to physical groups, for example, sonars 0 and 2 from logical group 0 are assigned to physical group (driver/amplifier board) 1.

VII. MML-11 SOFTWARE ARCHITECTURE

A. OVERVIEW

The Model-based Mobile-robot Language (MML) is a library of mobile robot-oriented functions written in the C language in UNIX environment [25]. It has been implemented on the autonomous mobile robot Yamabico-11 at UCSB and NPS. Over the years, MML has been improved through its many different versions. Currently, we are developing its eleventh version called MML-11.

From the robot control point of view, MML-11 is a programmable software system for mobile robot operation. The main procedure of the system conducts all necessary initializations for both hardware and software. A user program will be called after the initializations are done. Besides the main procedure, MML-11 mainly consists of two subsystems:

- Motion Control Subsystem and
- Sonar Control Subsystem

A user application program is required for the robot's operation using this software. For user application programming convenience, the system provides a set of well-defined functions called *user functions* as interface between the user and the system. The user functions are categorized into four modules:

- Operating System Module
- Motion Planning Module
- Motion Control Module
- Sonar Control Module

1. System Architecture

This software is developed with a special architecture which incorporates sequential structure and interrupt-driven structure. The system initialization and the user application program will be basically executed sequentially in the main procedure of the

system. Alternatively, the separate subsystems, i.e. Motion Control Subsystem and Sonar Control Subsystem, are periodically called for execution via interrupt requests for the required motion control and/or sonar control operation. Figure 7.1 illustrates the MML-11 software conceptual architecture.

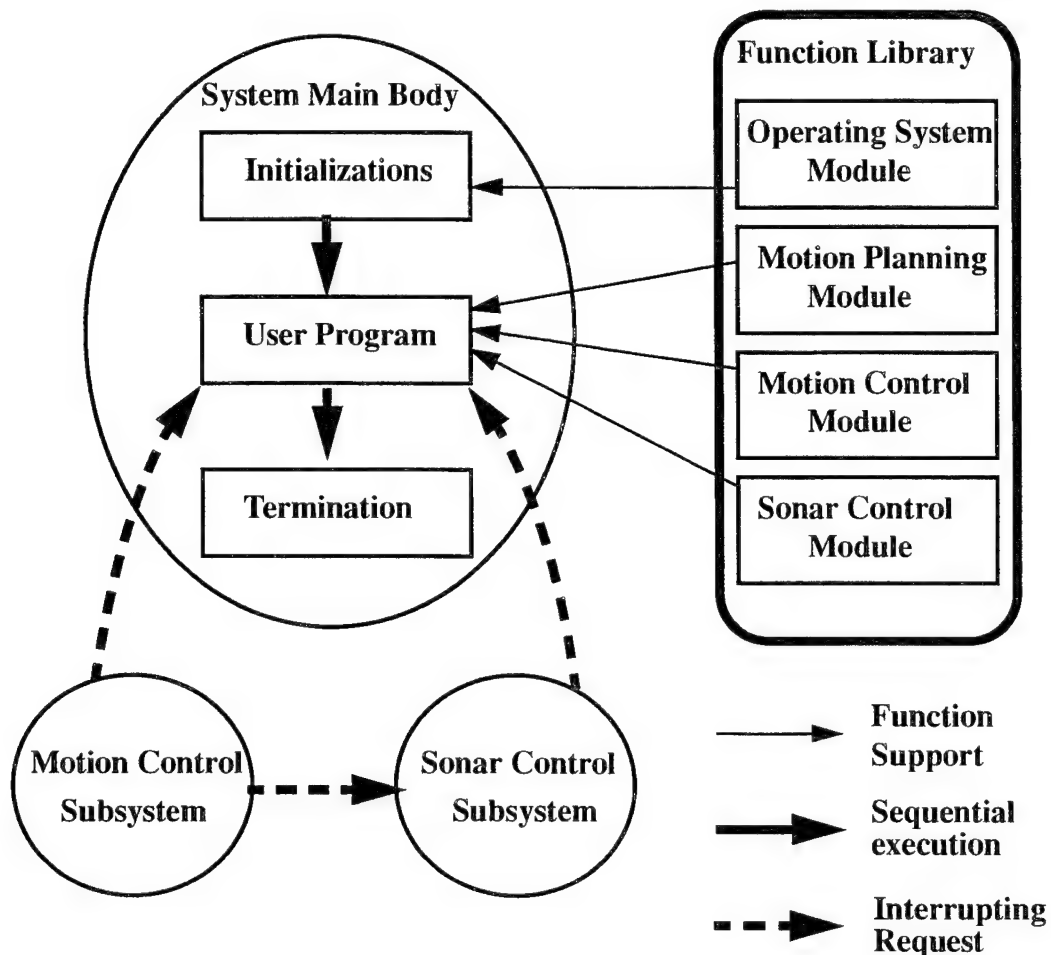


Figure 7.1: MML-11 Software Conceptual Architecture.

2. Interrupt-driven Subsystems

There are three primary tasks that may be running at any given time. The highest priority task is motion control subsystem, which performs all motion control computations

and in turn translates them into low-level wheels control. This subsystem is designed to interrupt other tasks in the frequency of 10 milliseconds. The next highest priority is the sonar control subsystem, which processes all incoming sonar returns and generates line segments from individual sonar returns from obstacles if required. The sonar control subsystem issues interrupt request in the frequency of 50 milliseconds. The lowest level priority, but basic, task is the user program. This part of the system feeds both immediate and sequential commands to the motion control subsystem through a command queue. All higher priority tasks interrupt the tasks with lower priorities to gain the CPU control. The design of MML-11 subsystems will be described in following sections.

3. Real Time Operating System

The Yamabico-11 onboard CPU, IV-SPARC 33, provides no standard operating system functions but a small set of libraries for console I/O. All other operating system primitives, such as interrupt handling, memory management, data formatting and logging have to be provided by the MML system. The detailed design of real time operating system for robotic system will be presented in Chapter VIII.

4. User Program

In this software, the robot's motion is instructed by the user program, which sends commands to the motion control system and/or sonar control system. However, motion planning and control specific concepts are hidden from the user. Only those defined as user functions are allowed to be used in user program. Sonar data is available to the user in either a raw or processed format via user sonar functions. Appendix C gives an user program example. The MML-11 user function specifications will be described in Chapter IX.

B. MOTION CONTROL ARCHITECTURE

The motion control must be performed in a short period repeatedly. It is difficult to impose this control in user's program. As we design interrupt-driven software system, the foreground job and background job concepts are introduced into MML-11 motion control

software. In MML-11, motion control mechanism is designed in the way that the execution of user program is somewhat separated from motion control. This allows users being able to program their applications by using simple functions. The user program is considered the *foreground process* which sends either immediate or sequential commands to the system. The robot motion control task conducted by motion control subsystem is considered the *background process* which performs motion control to achieve the motion instruction it gains control at a frequency of 10 ms. The immediate commands in the user program will be executed immediately, while the sequential commands will be enqueued to a buffer called the *instruction buffer* waiting for execution sequentially. The motion control subsystem fetches an instruction sequentially. When the execution of one instruction is finished, the control subsystem picks and executes another instruction from the buffer until the buffer is empty. The motion control architecture is illustrated in Figure 7.2.

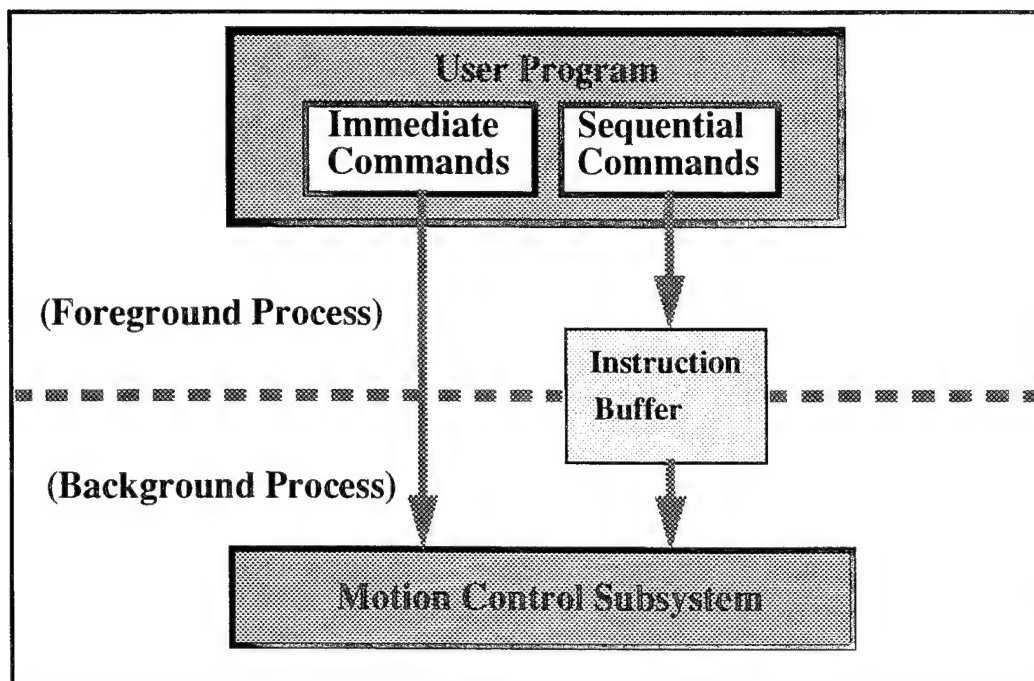


Figure 7.2: MML-11 Motion Control Software Architecture.

C. MOTION CONTROL SUBSYSTEM

Motion control subsystem, named **MotionSysControl**, is the foreground process of the entire system. It is designed to compute all data necessary for motion control by interrupting system main procedure (or user program) every 10 milliseconds. When the interrupt request is granted, this subsystem gains the control of CPU. It actually acts as an interrupt service routine. This section presents the structure of MotionSysControl subsystem and discusses how the robot decides its transitioning dynamically.

1. System Structure

MotionSysControl performs following computations for the robot motion control in order to accomplish its mission.

- Measure the distance traveled, Δs , in a cycle by reading robot's left and right shaft encoders.
- Compute the orientation changed $\Delta\theta$
- Localize current configuration q .
- Compute commanded linear and rotational velocity, V_L , V_ω , for next cycle
- Translate commanded velocity into control signals, *PWM*, for driving motors
- Transition point simulation to decide whether to read next instruction

The block diagram in Figure 7.3 illustrates motion control subsystem structure. By reading robot's left and right shaft encoders, the process can measure the distance traveled. Computations of distance traveled and orientation changed are done in order by a module with outputs Δs and $\Delta\theta$. These data will be used by localization module to compute robot's current configuration. The current configuration q is needed for motion rule module to compute commanded linear and rotational wheel velocities, V_L , V_ω , for next cycle. These velocities are translated in left and right PWMs as signals to drive corresponding motors. The last step in MotionSysControl is to determine whether or not to start transitioning to next path. This is done by a forerunner simulation in real time. If it decides to transition, the next motion command in the instruction buffer will be read and followed.

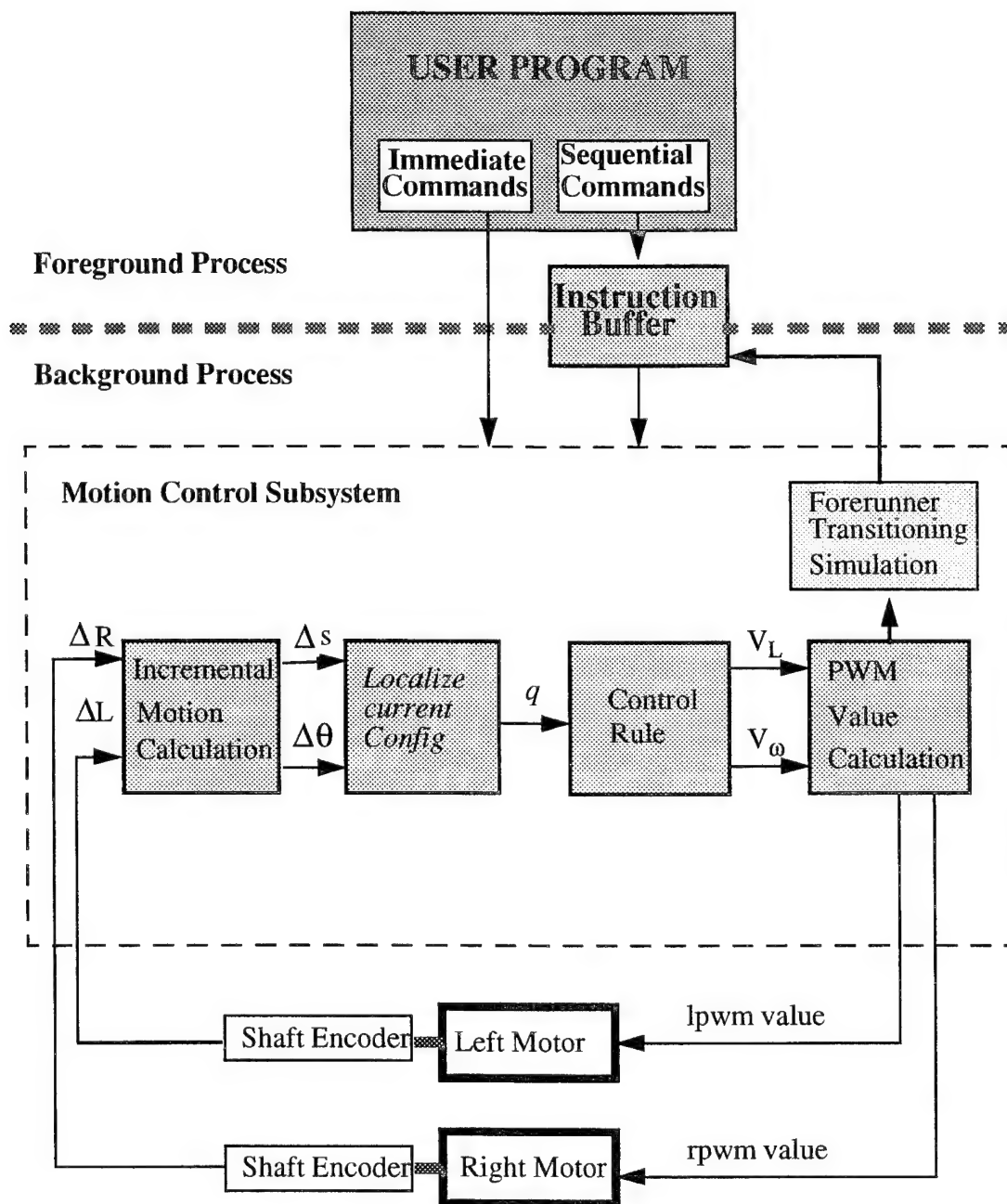


Figure 7.3: MML-11 Motion Control Subsystem Structure.

2. Transition from One Path to Another

The last step in MotionSysControl is to determine whether or not to start transitioning to next path. In MML-11 software system, the robot's motion can be controlled by tracking successive path segments using path tracking algorithm. The possible path segments are straight line and circular arcs. In order to smooth motion and to ensure safety, the robot must transition from one path to another at the right time and right position. The position where robot begins to transfer from the current path to the successive one is called *leaving point*. The *transition point* is defined as the last leaving point on the current path segment such that the robot's motion trajectory does not intersect the second path segment. The distance between the transition point and the intersection of two successive paths is called *transitioning distance*. Figure 7.4 shows the different trajectories of a robot transitioning from a path p1 to p2 at different leaving points. The Figure 7.4 (a) is an early leaving point which makes the trajectory longer to converge to path p2. In Figure 7.4 (b), the robot transitions to path p2 at proper leaving point. Figure 7.4 (c) shows a late leaving point making the trajectory intersect path p2.

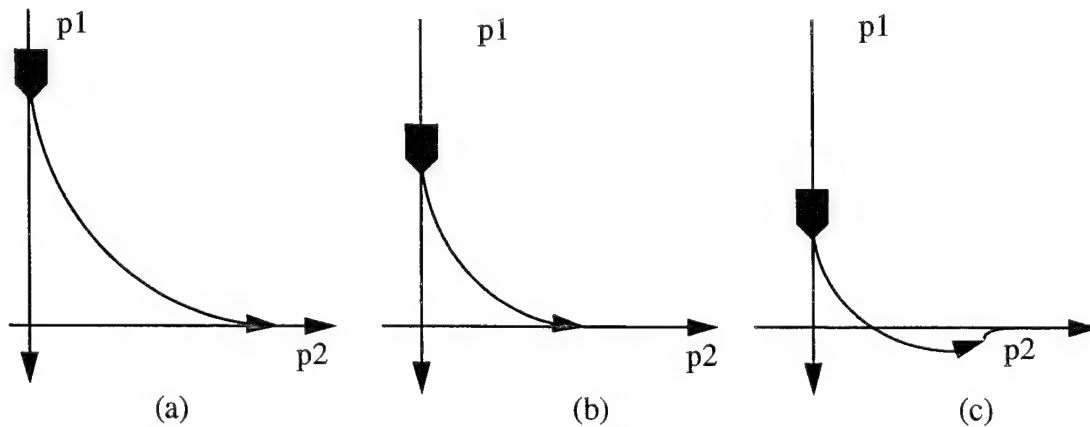


Figure 7.4: The Different Motion Trajectories

Therefore, transitioning from one path to another at the right time and right position is very much important in motion control. In this section, we introduce a new method to efficiently compute a transition point for robot transition control.

a. Real Time Dynamic Transition Control

In normal transition control, the transition point can be approximately calculated in foreground process when more than one path segments are present in user program. For instance for transitioning from a path segment to a perpendicular path segment, the transition distance is about twice as much as the given smoothness σ . Its transition point can be easily calculated according to this distance. After the transition point is obtained, the transition can be controlled by checking robot's current position against the transition point. When the robot reaches the transition point, it transitions to track the next path segment. Knowing transition distance, the transition point calculation is not too hard in this case. However, the transition distance is not fixed in other cases. On the contrary, it varies from case to case depending on the some other factors, e. g. orientation difference of two path segment, curvature limitations, velocity and so forth. This makes transition point calculation inaccurate. It requires complex functions or a large data table to properly determine the correct transitioning position.

For more flexible transition control, we propose a new method to determine when to transition dynamically without bearing heavy cost. The method is to use the *forerunner simulation* in real time. A forerunner is a virtual robot which is located in certain distance ahead of the real robot to simulate the robot's future trajectory. If the forerunner's trajectory converges to the reference path, another new forerunner is created based on robot's new configuration. These forerunner simulations keep running until the forerunner's trajectory intersects the reference path. When intersecting happens, it implies that transitioning from the initial position of this forerunner might be too late for a smooth motion. The correct transition point must be earlier than the last one. Therefore, we take middle position of the last two forerunners' initial position as the transition point.

Figure 7.5 illustrates forerunner simulation operation. In the figure, $p1$ represents current path and $p2$ represents successive reference path. When the real robot is in configuration $v1$, the forerunner, with distance S ahead of robot's current configuration, in configuration $f1$ is created. The forerunner simulates the robot's trajectory assuming that $f1$ is the transition point. When the first forerunner finished its job, the robot had moved to configuration $v2$ in Figure 7.5 (b). Since the trajectory of forerunner in $f1$ converges to path $p2$, the second forerunner in $f2$ is created with the same manner. The simulation of forerunner in $f2$ found that its trajectory intersects path $p2$. Thus the simulation stops and the transition point is computed as the point Tp in Figure 7.5(b).

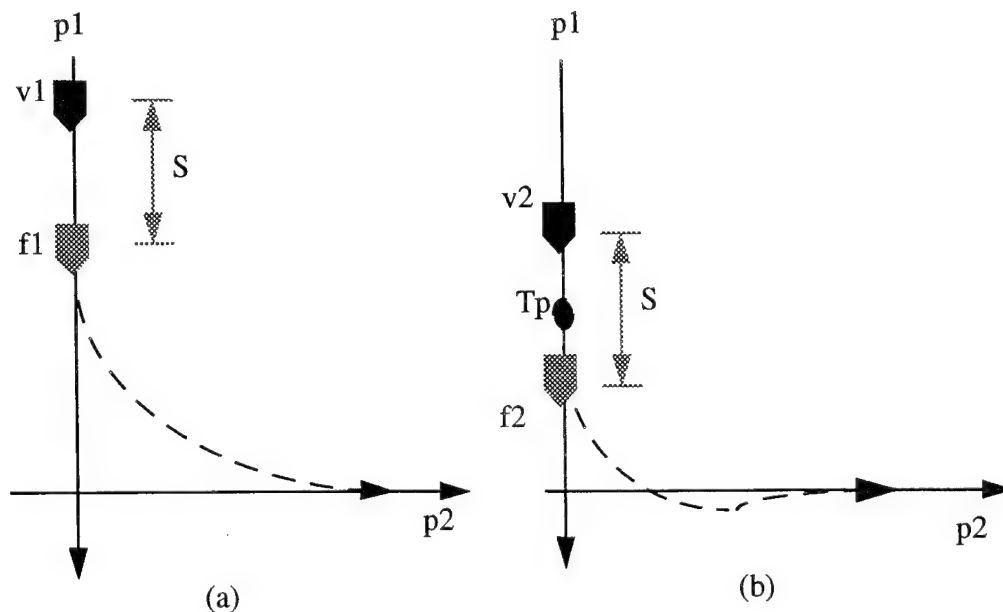


Figure 7.5: The Forerunner Simulation.

b. Fast Convergence Detection

In straight line tracking simulation, if the trajectory intersects the reference line, it can be identified easily by transferring the reference line to X axis (it means the line

is $y=0$) and checking Y coordinate of the trajectory. When Y coordinate becomes negative, it intersects the reference line. However, the simulation takes significantly longer time to identify the convergency situation if it is done with this method. Therefore a faster way to determine convergency of simulation is needed in forerunner simulation.

In this section, we are introducing a fast way to check the convergence situation in the line tracking. We define *shadow*, denoted ξ , as the distance between p_1 and p_2 , where p_1 is the image of a configuration $q = (p, \theta, k)$ on the simulated trajectory, and p_2 is the intersection of reference path and the tangent line of the trajectory at configuration q (see Figure 7.6).

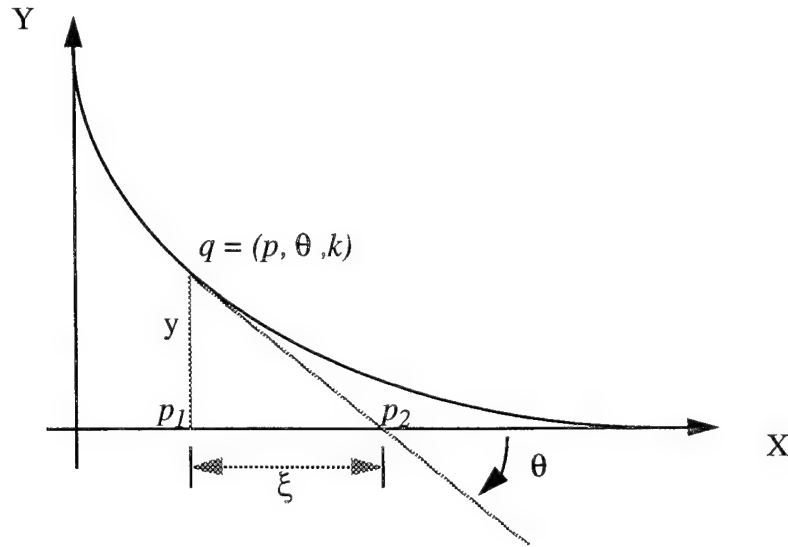


Figure 7.6: The Shadow of a Trajectory

It's obvious that the shadow ξ is geometrically related to the component of configuration q as equation 7.1.

$$\xi = \frac{y}{\tan \theta} \quad (\text{Eq 7.1})$$

We define the *normalized shadow*, denoted ξ^* , as equation 1.2, where σ is the given smoothness.

$$\xi_s^* = \frac{\xi}{\sigma} \quad (\text{Eq 7.2})$$

We observed the normalized shadows in the simulation, and found that the normalized shadow ξ_s^* reaches a maximum value in all cases. Figure 7.7.(a) shows the trajectory of tracking the line of $q = ((0, 0), 0, 0)$ with initial configuration $q_{init} = ((0, 10), -\pi/2, 0)$. Figure 7.7.(b) shows its positive normalized shadows vs. trajectory length. The maximum normalized shadow is $\xi_s^* = 0.18$ when the trajectory intersects the reference line. Another simulation result is shown in Figure 7.8. Figure 6.8(a) shows the trajectory of tracking the line of $q = ((0, 0), 0, 0)$ with initial configuration $q_{init} = ((0, 22), -\pi/2, 0)$. Figure 7.8(b) shows its positive normalized shadows vs. trajectory length. The maximum normalized shadow in this case is $\xi_s^* = 1.17$.

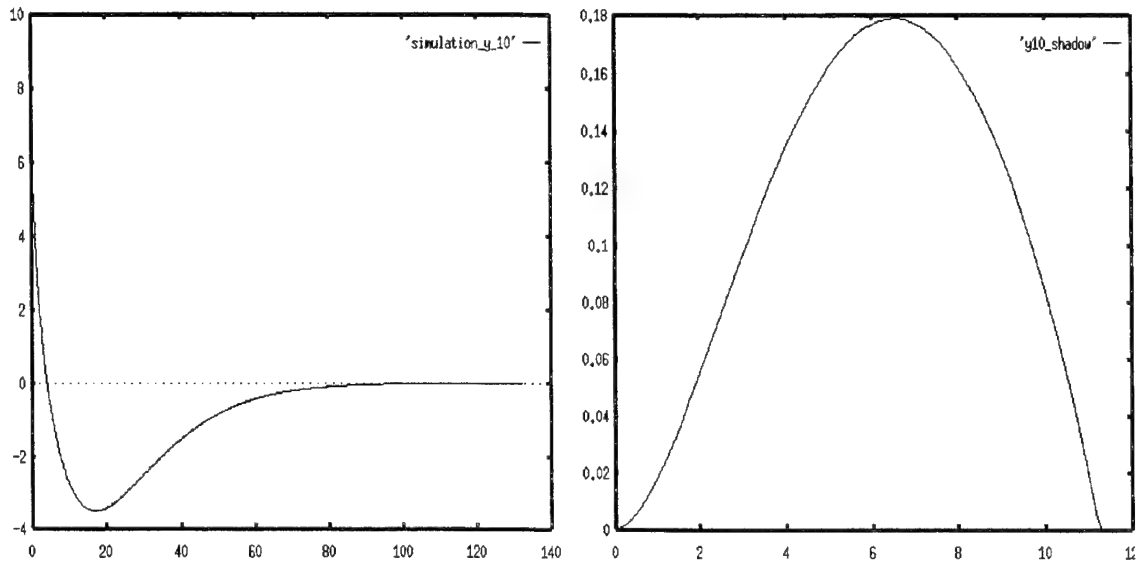


Figure 7.7: Positive Normalized Shadow vs. Trajectory Length I

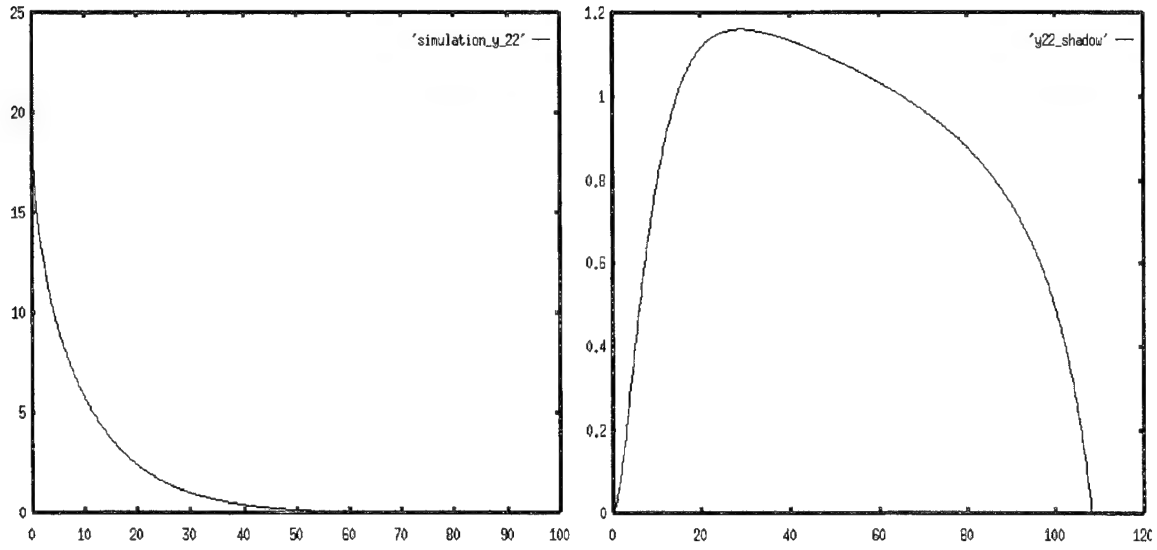


Figure 7.8: Positive Normalized Shadow vs. Trajectory Length II

The simulation results reveal an interesting phenomenon, in which $\xi_{\max}^* < 0.9$ if the trajectory crosses the reference path and if the trajectory converges to the reference path, $\xi_{\max}^* \geq 0.9$. The table 7.1 shows the experimental data with initial conditions given as follows.

- Initial configuration $q1 = (p, \theta, k) = ((0, y_0), -\pi/2, 0)$
- Reference path $q2 = (p, \theta, k) = ((0, 0), 0, 0)$
- Distance constant, $\sigma = 10.0$
- Step size, $\Delta s = 0.2$

Table 7.1: Max Shadow of Different Initial Configuration

y_0	10	18	19	20	22	25
ξ_{\max}^*	0.179	0.738	0.839	0.946	1.152	1.362
convergency	crossing	crossing	crossing	converging	converging	converging

The data in Table 7.1 shows the experimental results for some fixed initial condition. This phenomenon, however, is consistent with other given initial configuration and smoothness. Therefore, the fast way to determine whether a path tracking converges to its reference line is by checking the normalized shadow as:

- converge if $\xi^*_{\max} \geq 0.9$
- intersect if $\xi^*_{\max} < 0.9$

The fast converge detection method is especially valuable in real time dynamic transition control because the transition point can be determined earlier so that the transition motion can be much smoother. This method has been successfully applied to MML-11 which makes the software more flexible.

D. SUMMARY

MML-11 is the newest version in MML development history. One of the important features of this new version is that it is written in ANSI C, yet is designed with an object oriented perspective. Individual source files are similar to C++ classes by way of their data encapsulation and initialization. Each source file encapsulates all local data statically, and the only means of accessing the data from outside the file is through a well-defined set of interface functions [19]. This feature secures the important control data from being accessed or modified illegally resulting unexpected behavior. On software engineering point of view, this is especially helpful in the long term system development and maintenance.

VIII. ROBOT REAL TIME OPERATING SYSTEM DESIGN

A. INTRODUCTION

Operating systems are primarily resource managers. The main resource they manage is computer hardware in the form of processors, storage, input/output devices, communication devices and data. For a robotic system, the computing device is normally a commercial product in which the basic operating system embedded. For instance, Yamabico-11 is currently equipped with an Ironics IV-SPARC-33 CPU which is a single processor [26]. It provides general computing capabilities. In order to support robot control in various aspects, a suitable operating system needs to be developed. In this chapter we address the design of real time operating system for the robot focusing on tasks scheduling and dynamic memory management.

The design of robotic operating system mainly involves the robot hardware control. In what manner the attached hardware is to be controlled is the guideline of such a design. Robot motion control is the central issue of robotic system. How the operating system is designed to handle motion control is the most important of concerns. While the motion control is been carrying out, gathering surrounding information from the desired sensors for navigation is a natural operation. The robot's motion must react to the sensors information in most cases. For a single processor robotic system as Yamabico-11, the operating system is designed to determine when and how each of control routines takes over the CPU. Obviously, a multitasking operating system is needed for a robotic system to handle those controls. The multitasking operation is made possible by implementing an interrupt mechanism for the tasks. In addition to the tasks scheduling, dynamic memory management is another issue to be solved in Yamabico-11.

B. MULTITASKING OPERATING SYSTEM DESIGN

In order to properly design an interrupt-driven operating system for Yamabico-11, the interrupt mechanism of IV-SPARC-33 board must be understood. The IV-SPARC-33

board uses seven of 15 SPARC asynchronous traps to emulate 680x0 interrupt level 1 through 7 exceptions. The remaining eight asynchronous traps are undefined and are not presented to the SPARC processor. Table 8.1 lists the traps recognized by the IV-SPARC-33 board.

Table 8.1: SPARC/680X0 Interrupt Level Equivalence

SPARC Interrupt Level	680x0 Interrupt	IACK Address
1	1	0xFFFFFFFF3
2	2	0xFFFFFFFFE7
3	3	0xFFFFFFFFF7
4	4	0xFFFFFFFFEB
5	5	0xFFFFFFFFFB
6	6	0xFFFFFFFFEF
7 - 14	Spurious Interrupt	Undefined
15	7 (NMI)	0xFFFFFFFFF

Traps are controlled by several registers in SPARC board depending on the trap types. The interrupting traps are controlled by a combination of the Processor Interrupt Level (PIL) field and Trap Enable (ET) field of Processor Status Register (PSR). The ET field in the PSR must be 1 for traps to occur normally. While ET = 1, the Integer Unit (IU) prioritizes the outstanding exceptions and interrupt requests according to their priority ranks. For interrupt requests, the higher interrupt level possesses higher priority, e.g. interrupt level 2 has greater priority than interrupt 1, and so on. When an interrupt request occurs, the IU compares the Interrupt Request Level (IRL) against the PIL. If $IRL > PIL$ or $IRL = 15$ (Non-Maskable Interrupt, NMI), the processor takes the interrupt request trap and the control goes to that interrupt service routine [27].

Although the SPARC architecture specifies 15 interrupt levels, the onboard logic is capable of requesting only interrupt levels 1 through 6 and 15. Therefore, we must be aware of the interrupt-acknowledgment precautions that the interrupt level 7 through level 14 are not defined in SPARC board and interrupt level 15 (NMI) is a special case which requires a special handling. To avoid unexpected results, when we design interrupt-driven operating system, level 7 to level 14 should not be used in any case and the use of interrupt level 15 must be very carefully handled.

1. System Design Considerations

For a real-time robot system, the robot's motion and sensing devices operation should be controlled by the software at all time until its mission is accomplished. This implies that the interrupts of motion control system and sonar control system of Yamabico-11 should occur periodically. Thus, the determination of interrupt frequencies is an important part of the design. Another aspect of the design consideration is which task should possess higher priority so that it is allowed to interrupt other less important tasks and preventing being interrupted by them. The hardware interface which is available for the implementation of interrupt design should also be selected.

With these requirements, the frame work of multitasking operating system design is clarified. The system design may involve several decisions and implementations including:

- Priorities and frequencies assignment decisions;
- Selection of available interfaces;
- Interrupt handling service routines design and
- Installation of interrupt handler.

2. Interrupt Level and Frequency Assignment

Higher priority tasks are allowed to interrupt one or more lower priority tasks when required. Therefore, tasks should be assigned an appropriate priority depending upon their relative importances when designing an interrupt-driven operating system for a robotic

system. The higher the interrupt level is, the higher the priority of the associated task will be. Since the motion control is the most important task compared with other tasks, it should be assigned with the highest priority among the asynchronous trap level. The sonar control task can be assigned with other lower priority level. As Table 10.1 indicates, the highest interrupt level in SPARC board is Level 6 and the lowest level is 1. We design the operating system of Yamabico-11 with following priorities assignments:

- Motion control system: interrupt level 6.
- Sonar control system: interrupt level 3.

This design allows control of future sensing devices like vision system to be inserted between those levels according to their relative importance. The Figure 8.1 illustrates the interrupts which may occur in the MML-11.

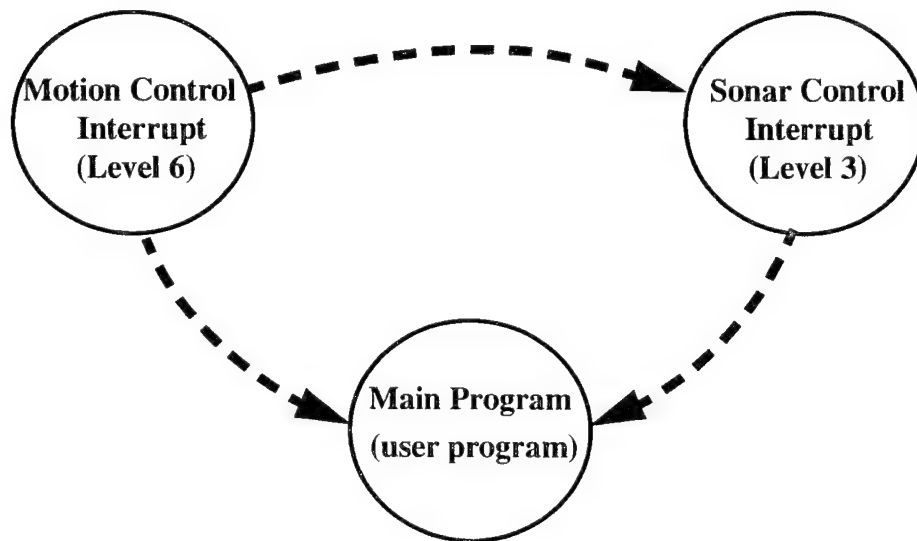


Figure 8.1: MML-11 Interrupts.

In addition to the priority assignment, the frequencies of interrupts also needs to be determined in the design. For more precise control of robot's motion, motion control system interrupts should occur over short periods, while still allowing other task to regain control. Thus, we design the frequency for motion system control to be 100 Hz, and for

sonar system control 20 Hz. This means the motion system control interrupt occurs every 10 milliseconds, while the sonar system control interrupt occurs every 50 milliseconds. And that the motion system control task will interrupt the sonar system control. In our estimation, the motion system control task runs for approximately 400 microseconds (0.4 ms) and sonar control system task runs for approximately 3200 microseconds (3.2 ms).

3. Selection of Available Interface

The programmed interrupts are made possible by specifying desired interrupts using onboard registers as an interface. Some of the unused board configuration registers are chosen for our interrupts design as follows:

a. Registers for Motion System Control

We use Timer 1 Interrupt for the interface to enforce motion system control. The Timer 1 can be set to interrupt at rates of 50 Hz, 100 Hz, or 1000 Hz. This interrupt is controlled by:

- Timer 1 Interrupt Control Register (address 0xFFFC002B)
- Local Interrupt Vector Base Register (address 0xFFFC0057)
- Slave Select 0 / Timer 1 Control Register (address 0xFFFC00C3)

The vector return to the CPU when it issues an interrupt acknowledge in response to the Timer 1 interrupt is 0x4A. This vector indicates the location of interrupt handler vector table which will be replaced by the address of the interrupt service routine. Thus using a variable, **MotionIntVector**, to represent the vector location, we have:

MotionIntVector = 0x4A;

b. Registers for Sonar System Control

For sonar system control the VMEbus IRQ2* Interrupt Control Register (address 0xFFFC00B) is selected because the sonar card is connected to VMEbus line IRQ2*. The vector associated to VMEbus IRQ2* Interrupt Control Register is identified as 0xA2. Using another variable, **SonarIntVector**, to represent this vector location, we have:

SonarIntVector = 0xA2;

4. Interrupt Service Routine

The interrupt service routines can be executed upon receipt of the vector associated with a specific interrupt. The content of an interrupt service routine reflects its desired functionality. For motion system control, the routine, named **MotionSysControl**, is designed firstly to read the shaft encoders and computes the vehicle's odometry configuration estimate. Then the desired curvature and velocity are calculated. This information is used to determine the necessary pulse width modulation commands for controlling the left and right wheel drive motors. For the sonar system control, the routine, named **SonarSysControl**, calculates the range of enabled sonar from its returned value. Then write it to the memory in order to be used as required. The detailed design of routines is beyond this chapter.

5. Interrupt Handler Installation

The steps required to install an interrupt service routine are:

- Set up the required interrupt control registers
- Assign interrupt service routine vector location.

The design of motion control interrupt and sonar control interrupt follow these steps. They are described this the following two sections.

a. Motion System Control Interrupt Handler Installation

Firstly we set up the Timer 1 Interrupt Control Register (0xFFFC002B) which specify the interrupt level. The bit 7 of this register is used to enable the interrupt associated with the timer. Its value 0 indicates the interrupt is enabled. The bit 6:4 are reserved. Their values should be kept as they are in default setting. The bit 3 is read only, and a write on this bit has no effect. The bit 2:0 specify the interrupt level seen by the CPU. Setting this register to enable the motion system control interrupt in level 6 makes the contain of the register as [00110110]. This setting can be done by using bit-wise operation based on default value of the register. Two additional bit-wise variables are set as

InterruptMask = 0x30 and InterruptLevel6 = 0x06 to accomplish the logic operation as Figure 8.2.

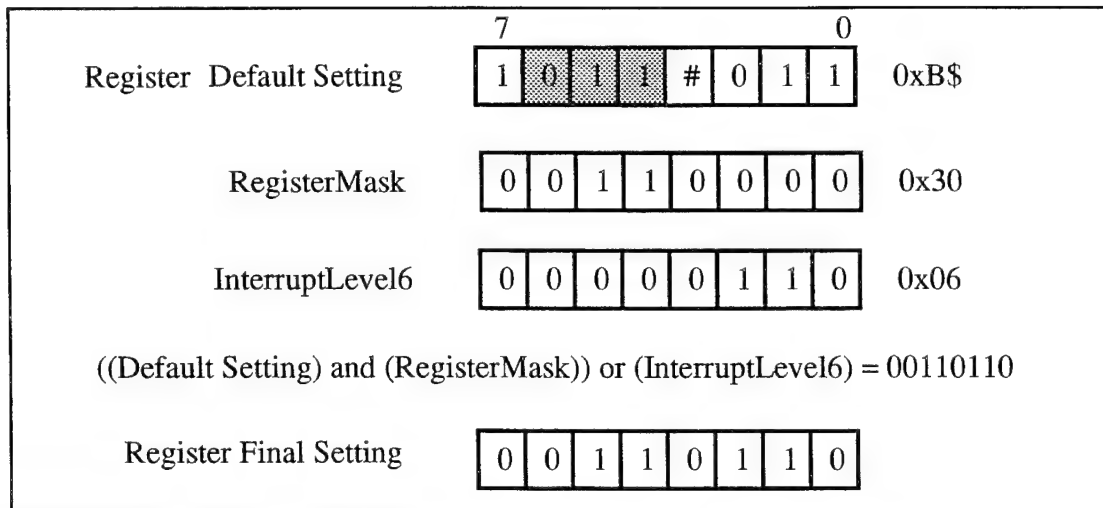


Figure 8.2: Setting Timer 1 Interrupt Control Register

Then set up the Slave Select 0 / Timer 1 Control Register (0xFFFC00C3) which is mainly used to specify the interrupt frequency. The setting is done by logic operation on register default setting and a bit-wise variable Enable100Hz = 0x80 as Figure 8.3. The final contain of this register is [11010010] where the value 11 in bits 7 and 6 set the frequency to 100 Hz.

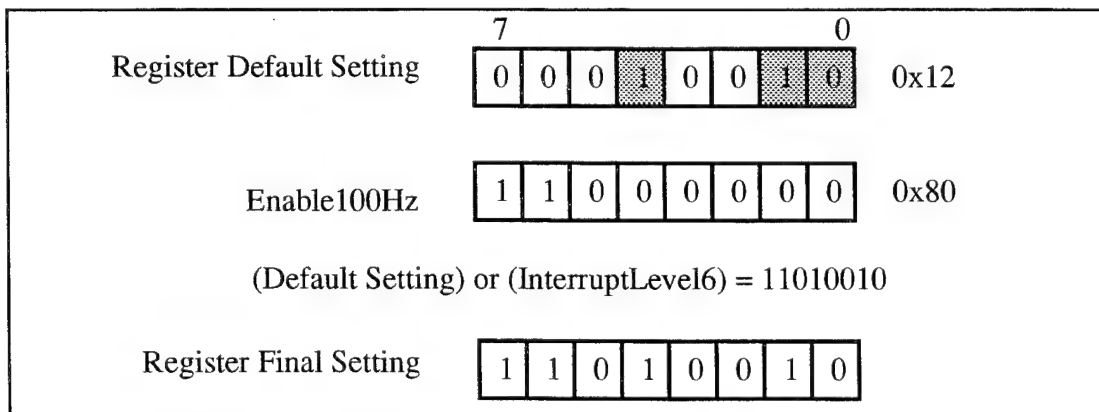


Figure 8.3: Setting Slave Select 0 / Timer 1 Control Register

After those registers are set up, the interrupts are ready to run. Then we can install the motion system control interrupt service routine address into the corresponding interrupt service routine vector location, using the **mk_handler()** library function as below:

```
mk_handler(MotionIntVector, *MotionSysControl);
```

MotionIntVector: Location in interrupt handler vector table.

*MotionSysContro: Function pointer to be placed into handler vector table;

b. Sonar System Control Interrupt Handler Installation

Sonar system control interrupt frequency is set by firmware on the sonar card. The implementation of sonar interrupt involves setting the VMEbus IRQ* interrupt Control Register only. In eight-bit width register, the bit 7 is used to enable the interrupt. Its value 0 indicates the interrupt is enabled. The bits 2:0 specify the interrupt level seen by the CPU. The rest of bits are reserved and should be kept as they are in default setting. To set the sonar system control interrupt to level 3 and enable the interrupt, bit-wise logic operations are performed on the register with its default setting and two other variables, RegisterMask = 0x78 and InterruptLevel3 = 0x03. The final setting on the register is [01111011] as Figure 8.4. The values in the bit 2:0 indicates the sonar interrupt level being set to level 3.

		7						0	
Register Default Setting		1	1	1	1	1	1	0	1
									0xFD
RegisterMask		0	1	1	1	1	0	0	0
									0x78
InterruptLevel3		0	0	0	0	0	0	1	1
									0x03
((Default Setting) and (RegisterMask)) or (InterruptLevel3) = 01111011									
Register Final Setting		0	1	1	1	1	0	1	1

Figure 8.4: Setting VMEbus IRQ* Interrupt Control Register

As motion system control interrupt handler installation, we can install the sonar system control interrupt service routine address into the corresponding interrupt service routine vector location, using the **mk_handler()** library function as below:

```
mk_handler(SonarIntVector, *SonarSysControl);
```

SonarIntVector: Location in interrupt handler vector table.

*SonarSysContro: Function pointer to be placed into handler vector table;

C. DYNAMIC MEMORY MANAGER DESIGN

Time Complexity and space complexity have been two important measurement of an algorithm in evaluation of efficiency. That is why the dynamic memory management is so important in a software development. MML-11 is a set of robot control functions written in ANSI C. The C language has already had memory allocation/deallocation functions to manage dynamic memory request. Then why bother to install them in MML-11? The answer is because Yamabico-11 is a self-contained robot. There is no operating system in its CPU -- IV-SPARC 33 to handle the memory management. In order to use the available memory in SPARC 33 economically and efficiently, MML-11 has to have its own memory management capability.

The most common known methods used in dynamic memory management are the following buddy systems [28]:

- Binary buddy system.
- Fibonacci buddy system.
- Boundary tag buddy system.

We adopt **Fibonacci buddy system** for use in MML-11 for the following two reasons. Firstly it allows for a greater variety of possible block sizes in a given amount of memory than Binary buddy system. Second, the Fibonacci buddy system provides an efficient method to allocate and deallocate memory blocks. The techniques the Fibonacci buddy system used to allocate and deallocate memory are briefly described in Appendix B.

In order to properly design a memory management system, the memory organization of working CPU board needs to be understood.

1. IV-SPARC 33 Memory Organization Feature

Yamabico-11 onboard CPU has DRAM of size 16 Mega bytes. Its memory organization is illustrated in Figure 6.1. The DRAM array is organized into two-word (64-bit) interleaved banks of 8 Mbytes each. Bidirectional latching data buffers direct to and from the two banks. This is the physical memory space where we can load the robot software for execution. Loading target process to the memory is in upward direction starting from the bottom. Although DRAM address ranges from 0x00000000 to 0x00FFFFFF, the space between 0x00000000 and 0x00018000 is reserved for system use. Thus, when a target process (the compiled programs called kernel) is downloaded to the SPARC board, it starts at address 0x0001800. Following the kernel, the uninitialized variables are placed followed by the initialized variables. The top address of the memory block is occupied by the target process and its variables depends on the size of the programs. On the upper part, the space between 0x00FFFFFF and 0x00FFFF00 is reserved for future use. The rest of memory space in DRAM will be used for stack of the client's program. The memory used by the stack is arranged in downward direction from the top the DRAM (actually it starts at the address 0x00FFFF00).

2. Memory Block for Dynamic Allocation

When allocating a memory block to a request dynamically, caution must be taken that the memory space designated to system operation should not be interfered with. For SPARC-33 CPU, it's obvious that the only space that can be used for this purpose is the space from the bottom of program stack down to the top of memory occupied by initialized variables. Unfortunately, those addresses are not fixed. In other words, it is difficult to exactly measure what space will be available for dynamic allocation use. However, the way CPU handle the DRAM helps us find the appropriate space. As aforementioned, the target process stack will consume the space from the address 0x00FFFF00 downward. In the

meantime, target process text (program) and data are stored upward starting from 0x00018000. In order to prevent possible overlaps between the data in dynamic memory blocks and that of other spaces, our policy is to set the dynamic memory block of size 4 Mbytes in the middle of DRAM. This decision of choosing proper space for use is reasonable. By looking at the size of object codes of current MML-11 program, we found that it is relatively small (about 400 kbytes) compared to the total size of DRAM in SPARC. Therefore we decide to make the top address of dynamic memory space at 0x00800000 which is exactly the midway point of 16 Mbytes DRAM. This allows for a stack having about 8 Mbytes to consume and the MML-11 kernel can be extended up to more than 3 Mbytes.

3. Memory Block Partition

In order to allow a memory block being returned to the available list to be coalesced with any other block(s) in the list, the memory block must be partitioned in such a way that any size can be represented as the sum of two smaller sizes and that the neighbor blocks are easy to identified. The most famous sequence of numbers having this property is the Fibonacci sequence. Therefore, the Fibonacci buddy system is adopted for memory block partition in MML-11 operating system design (see Appendix B). We design dynamic memory allocation function with memory block partitioned into the sizes of Fibonacci sequence (in byte) as follows:

$$F_1 = 8$$

$$F_2 = 16$$

$$F_i = F_{(i-1)} + F_{(i-2)} \quad \text{for } i > 2$$

The sequence generated is as follows: <8, 16, 24, 40, 64, 104, 168, 272, 440, 712, 1152, 1864, 3016, 4880, 7896, 12776, 20672, 33448, 54120, 87568, 141688, 229256, 370944, 600200, 971144, 1571344, 2542488, 4113832>.

Why do we set $F_1 = 8$ instead of any number smaller? This is because IV-SPARC-33 memory has 8 bytes addressable boundaries and this size of the block will be used to

compute the address of its buddy blocks. Thus all partitioned blocks must be of size of multiple of 8 to meet address alignment requirement. This important feature was identified after many testing.

4. Dynamic Memory Allocation Functions Design

As stated in Appendix B, the bookkeeping data is needed in each block for Fibonacci buddy system operation. In MML-11, we define a data type in C as follows:

```
typedef struct head{  
    unsigned int  Active:1;  
    unsigned int  Lbc:7;  
    unsigned int  Size;  
    struct head  *next;  
    struct head  *prev;  
} HEADER;
```

where Active is used to indicate whether the block is active or not. The unsigned integer Lbc is used to maintain a record of how deeply the block is nested as left buddy of other blocks. The size is obviously used to indicate the size of a memory block. The next and prev fields of the structure are pointers pointing to its next and previous block in the available list respectively. When a block is active, i.e. being assigned to a request, these two fields are not used. The algorithm for maintaining this left buddy count involves these steps.

- As a block is split, the resulting left buddy has its left buddy count field increased by one. The resulting right buddy has its left buddy count field set to zero.
- As coalescing occurs, the left buddy must always have its left buddy count field decreased by one.

With the data structure for bookkeeping data of the memory block, the memory allocation and deallocation functions can be designed using the general algorithms described in [28].

IX. MML-11 LANGUAGE SPECIFICATION

A. OVERVIEW

In this chapter, we describe the design of user functions which will be used as interface between user and MML-11 software. The specifications of functions for motion control, sonar control and geometric calculation are presented. Some of basic data structures which will be used to describe the functions are presented also. The user functions are categorized into following subsets:

- Geometric functions
- Motion planning functions
- Motion control functions including sequential functions and immediate functions.
- Sonar control functions.

The geometric functions defines some utility functions for the algebraic manipulation of geometric variables. The motion planning functions provide the user simple interface functions to build a world model and to conduct motion planning when giving a specific mission. The motion control functions include sequential functions and immediate functions. The sequential functions define a set of motion control commands that will be stored in a buffer when they are used in the user program and will be executed sequentially as robot's background tasks. The immediate functions define the commands which take effect immediately when they are executed in user's program. The sonar control functions are the functions used to control sonar operation and to obtain sonar data.

B. DATA STRUCTURE

1. Point

The POINT structure is used to describe a position in two-dimensional cartesian coordinates system. The structure includes a double X and a double Y.

2. Velocity

The VELOCITY structure is used to describe a two dimensional velocity vector. The data structure is made up of two doubles that represent the linear and rotational elements of velocity. They are appropriately named Linear and Rotational, respectively, in the VELOCITY structure.

3. Configuration

The CONFIGURATION is the standard structure for describing location and direction for an object. It consist of Posit, with type of POINT, which identifies an objects position in two dimensional cartesian coordinates. Another element is Theta of type double that describe's the object's orientation in relation to the X coordinate. Finally, there is another double called Kappa that represents the curvature of an object's path.

4. Path Element

The PATH_ELEMENT data structure is used to describe and store the various types of movements. This data structure consist of config which is of type CONFIGURATION. It holds the configuration of the path that the robot is to follow. PATH_ELEMENT also contains pathType, which is of type PATH_TYPE. A PATH_TYPE is a data structure used to identify the various paths that are available to the robot. It consist of the mode which is of type MODE and class which is of type CLASS. Type MODE is an enumeration type that gives a name to each path that the robot follows. Presently, the modes that are available include NOMODE, ENDMODE, STOPMODE, PATHMODE, ROTATEMODE, BIDIRMODE, KSPIRALMODE, PARAMODE, FOLLOWMODE and REGIONMODE. Type CLASS, which is also an enumeration type, is used to name and categorize the various PATHMODE types. The list of classes include NOCLASS, LINECLASS, CIRCLECLASS, BLINECLASS, and BIDIRCLASS.

C. USER FUNCTION SPECIFICATION

1. Geometric Functions

a. Euclidean Distance

Syntax: double **euDis**(*p1*, *p2*)

Parameters: POINT *p1*;
 POINT *p2*;

Description:

This function computes and returns the Euclidean distance between two given points

b. Normalize

Syntax: double **norm**(*angle*)

Parameters: double *angle*;

Description:

This function, when given an angle in radian, returns a normalized angle between $-\pi$ and π . This is the most common normalizing function used in the system.

c. Define Configuration

Syntax: CONFIGURATION **defineConfig**(*x*, *y*, *theta*, *kappa*)

Parameters: double *x*;
 double *y*;
 double *theta*;
 double *kappa*;

Description:

When passed the values that define a configuration (*x*, *y*, *theta*, *kappa*), this function allocates and assigns a configuration. It returns a configuration.

The configuration can be used to represent a path which is either a line or a circle.

If the configuration is defined with curvature zero, i.e. *kappa* = 0.0, it specifies a straight line passing through the point (*x*, *y*) with orientation *theta*. If its curvature

is greater than zero, i.e. $\kappa > 0.0$, the path is a counterclockwise circle. If $\kappa < 0.0$, then the path is a clockwise circle. Figure 9.1 illustrates these concepts

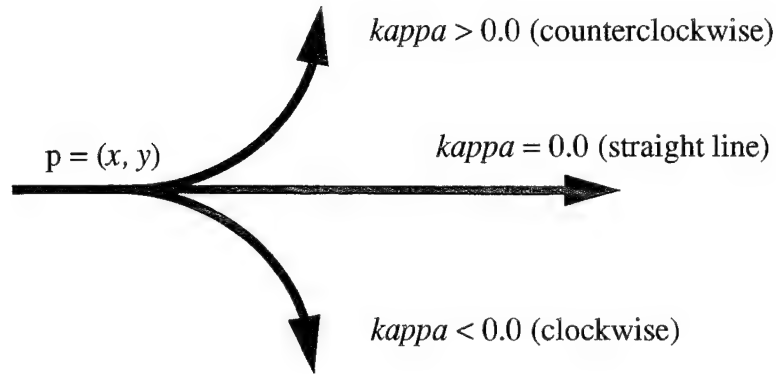


Figure 9.1: A Configuration Represents a Line or a Circle

d. Reverse Configuration

Syntax: CONFIGURATION **reverseConfig**(*original*)

Parameters: CONFIGURATION *original*;

Description:

The purpose of this function is to reverse the orientation of a given configuration by 180 degrees. It returns the reversed configuration.

e. Inverse

Syntax: CONFIGURATION **inverse**(*original*)

Parameters: CONFIGURATION *original*;

Description:

The purpose of this function is to calculate the inverse of a given configuration such that: *original* * inverse = Identity. The parameter *--original* is the original configuration in global coordinates. This function returns the inverse configuration.

f. Compose

Syntax: CONFIGURATION **compose**(*first*, *second*)

Parameters: CONFIGURATION *first*;
CONFIGURATION *second*;

Description:

The purpose of this function is to calculate the composition of two configurations. Specifically, the function takes parameter -- *first* and composes it with parameter - *second* to calculate and return the composed value. The returned value is the goal configuration in global coordinates.

g. Circular Arc

Syntax: CONFIGURATION **CircularArc**(*l*, *alpha*)

Parameters: double *l*;
double *alpha*;

Description:

Given a tangential orientation *alpha* and the arc length *l* in a curve, this function computes its configuration in the local coordinate system [19]. In motion control case, length would actually be delta-s and alpha would be delta-theta. The function can be called to determine the configuration after the incremental move in the local coordinate system of the original configuration.

h. Intersection of Two Lines

Syntax: CONFIGURATION **intersectLineToLine**(*q1*, *q2*)

Parameters: CONFIGURATION *q1*;
CONFIGURATION *q2*;

Description:

Given two configurations representing straight two lines, this function calculates the intersection of lines and returns the configuration of the intersection.

i. Depth

Syntax: double **depth**(*p*, *alpha*)

Parameters: POINT *p*;

double *alpha*;

Description:

Given a point *p* and the orientation of the point *alpha*, this function computes the depth of the point along a line defined by the parameters. It returns a value of type double.

2. Motion Planning Functions

a. Creating World Model

Syntax: void **createModel**(*world*)

Parameters: worldModel *world*;

Description:

This function creates a world model decomposing the free space of a given world into K-regions. It will generate the a set of data which is needed in planning robot's motion. The resultant data includes a world model, World, a region table, RegionTable, a border table, BorderTable, an edge table, EdgeTable and connectivity graph, CGraph.

b. Global Path Planning

Syntax: void **FindPath**(*world*, *startConfig*, *goalConfig*)

Parameters: worldModel *world*,

CONFIGURATION *startConfig*,

CONFIGURATION *goalConfig*;

Description:

This function finds the shortest path connecting the initial region and the final region which contains start configuration and goal configuration respectively. The

output of this function is a sequence which begins with a region followed by a border repeatedly and ends with a region. In the sequence, the first element is the region which contains start configuration and the last element is the region where the final configuration resides. The border between two regions in the sequence is their common.

c. Local Motion Planning

Syntax: void **LocalMP**(*world*, *startConfig*, *goalConfig*, *path*)

Parameters: worldModel *world*,
CONFIGURATION *startConfig*,
CONFIGURATION *goalConfig*;
pathClass *path*;

Description:

This function generates the motion instructions for each regions along the path. Those instructions will be taken to drive the robot in each region and finally stop the robot at the final configuration.

3. Motion Control Sequential Functions

The sequential functions define a set of motion control commands which are stored in a buffer that acts as an interface between user and robot. When the user program is being executed, commands of this type included in the user program do not take effect immediately instead they are loaded in buffer as motion instructions. The motion control system reads the instructions from the top of buffer sequentially and controls the robot's motion accordingly. The transition control from one instruction to another is described in Chapter IX. The specifications of those functions are listed below.

a. Tracking a Line

Syntax: void **line**(*q*)

Parameters: CONFIGURATION *q*;

Description:

The function defines a command that orders the robot to follow the line or circle specified by the configuration q . If the robot's last configuration before the command is executed is not on the track of the line specified, the robot uses steering function to transfer to the line with smooth motion. Figure 9.2 illustrates robot's behavior when executing $\text{line}(q)$ with a straight line q .

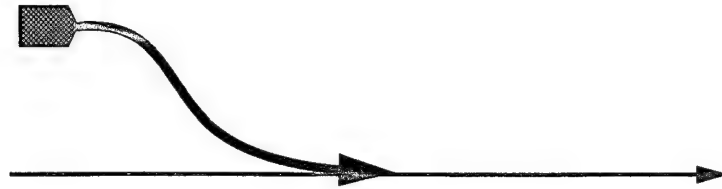


Figure 9.2: The Line Function

b. Tracking the Line from Its Back and Stopping

Syntax: void **stop0**(q)

Parameters: CONFIGURATION q ;

Description:

This function defines a command which steers the robot to track the line specified by the configuration q from its back. If the robot's image is on the back half of the line, the robot tracks the line as function $\text{line}()$ and stops when its image reaches the configuration. If the robot's image falls on the forward part of the line initially, the robot would not move. (see Figure 9.3)

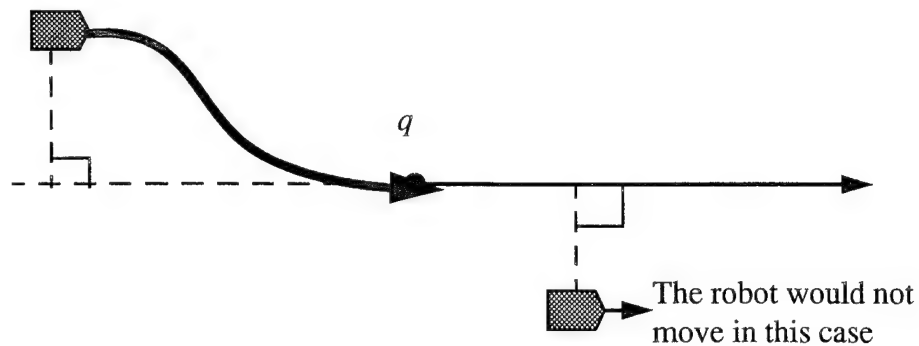


Figure 9.3: The Backward Line Tracking with Stop

c. Symmetrically Tracking a Line from Its Back with Stopping

Syntax: void **stop1**(q)

Parameters: CONFIGURATION q ;

Description:

This function defines a command which steers the robot to track the line specified by the configuration q from its back with a symmetric trajectory. This type of line tracking is called *symmetric line tracking*. Let q_s be the robot's initial configuration and q_{sr} be a line specified by the reversed configuration of q_s . Assuming q_r is the reversed configuration of the parameter q passed in by the function. In the symmetric line tracking, a forerunner (virtual robot) simulation is needed to generate a trajectory running back from q_r by tracking the reference line specified by q_{sr} . The following steps are taken in order:

- Step1: While the real robot is moving straight forward (by tracking the line specified by the initial configuration q_s), run the forerunner backward as stated above and store the trajectory in a path Π by appending a reverse configuration (negate its curvature also) of forerunner's current configuration.
- Step 2: Compare real robot's position against forerunner's. If they do not

meet, continue Step1. Otherwise, stop the forerunner simulation and start to track the prestored path Π until the robot reach the configuration q .

- Step 3. When the configuration q is reached, the robot stops.

Figure 9.4 illustrates the symmetric line tracking concept. In using this function, the caution must be taken that it is user's responsibility to make sure there is enough distance between initial configuration q_s and configuration q to allow forerunner's trajectory to converge to its reference line. Otherwise the real robot and forerunner may not meet each other and it may cause an unpredictable robot behavior.

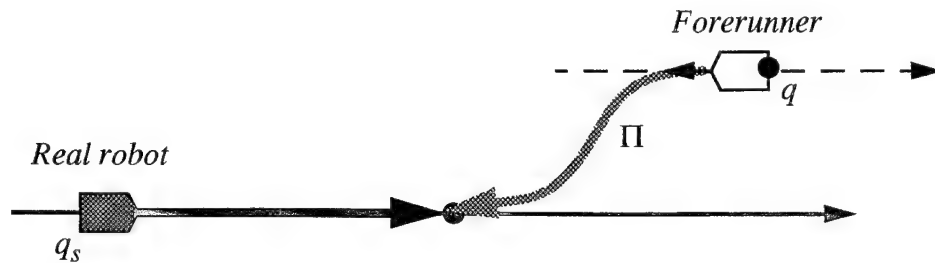


Figure 9.4: The symmetric Line Tracking with Stop

d. Symmetrically Tracking a Line from Its Back without Stopping

Syntax: void **pass**(q)

Parameters: CONFIGURATION q ;

Description:

This function defines a command which can make a robot have similar behavior as symmetric line tracking defined in function `stop1()` except it will not stop at the configuration q . Instead, when robot pass configuration q , it keeps tracking the line specified by q .

e. Rotation

Syntax: void **Rotate**(θ)

Parameters: double θ ;

Description:

This function is to command the robot to rotate a given angle. Negative angle makes the robot rotate clockwise, while positive angle makes an counterclockwise rotation.

f. Switch Robot's Heading Direction

Syntax: void **switchDir()**

Parameters: void;

Description:

This function reverses the heading direction of the robot.

g. Set Robot's Configuration

Syntax: void **setRobotConfig(*q*)**

Parameters: CONFIGURATION *q*;

Description:

This function sets robot's configuration to a given configuration *q*.

4. Motion Control Immediate Functions

a. Set Path Element

Syntax: void **setPathElement(*newPath*)**

Parameters: PATH_ELEMENT *newPath*;

Description:

This function sets the value of the current path element in motion control to the path element passed in as a parameter.

b. Get Path Element

Syntax: PATH_ELEMENT **getPathElement(*void*)**

Parameters: void;

Description:

This function retrieves the current path element in the motion control module.

c. Set Robot's Configuration Immediately

Syntax: void **setRobotConfigImm**(*q*)

Parameters: CONFIGURATION *q*;

Description:

This function sets robot's configuration to a given configuration *q* immediately.

d. Set Robot's Linear Speed Immediately

Syntax: void **setLinVelImm**(*speed*)

Parameters: double *speed*;

Description:

This function sets the robot's linear velocity immediately.

e. Set Robot's Rotational Speed Immediately

Syntax: void **setRotVelImm**(*speed*)

Parameters: double *speed*;

Description:

This function sets the robot's rotational velocity immediately.

f. Set Robot's Linear Acceleration Immediately

Syntax: void **setLinAccImm**(*acc*)

Parameters: double *acc*;

Description:

This function sets the robot's linear acceleration immediately.

g. Set Robot's Rotational Acceleration Immediately

Syntax: void **setRotAccImm**(*racc*)

Parameters: double *racc*;

Description:

This function sets robot's angular acceleration for speed changes in rotation.

h. Set Sigma Immediately

Syntax: void **setSigmaImm**(*sigma*)

Parameters: double *sigma*;

Description:

This function sets the robot's Sigma which control the sharpness of it trajectory when the robot is turning.

i. Set Total Distance traveled Immediately

Syntax: void **setTotalDistanceImm**(*distance*)

Parameters: double *distance*;

Description:

sets the total distance travelled by the robot to the value passed as a parameter

j. Get Total Distance traveled Immediately

Syntax: double **getTotalDistanceImm**(*void*)

Parameters: void;

Description:

Returns the total distance travelled by the robot.

k. Stop Immediately

Syntax: void **stopImm**(*void*)

Parameters: void

Description:

This function stops the robot immediately with the current acceleration rate until the speed reaches 0.

l. Halt the Robot Immediately

Syntax: void **haltImm**(*void*)

Parameters: void

Description:

The function brings the robot to a stop with the current acceleration rate. The robot stays in the state until `resumeImm()` function is called. When the `resumeImm()` function is called, it resumes the original motion without changes.

m. Resume the Robot's Motion Immediately

Syntax: void **resumeImm**(*void*)

Parameters: void

Description:

The function allows the robot to resume the motion it was executing before the `haltImm()` command was given.

n. Set the Robot's Motor On

Syntax: void **MotionOn**(*void*)

Parameters: void

Description:

Enables the motor control functionality.

o. Set the Robot's Motor Off

Syntax: void **MotionOn**(*void*)

Parameters: void

Description:

Disables the motor control functionality. By calling this function, users can push Yamabico as they like.

p. Logging Motion Data

Syntax: void **Motionlog**(**Filename, Frequency, BufferSize*)

Parameters: char **Filename*, int *Frequency*, int *BufferSize*;

Description:

This function prepares the tracing system to log motion data. Tracing is automatically turned on after this function is called. The *Filename* specifies a file

name that will be used to store data when the data is uploaded to the host. The *Frequency* specifies how many sonar cycles are skipped before data is logged.

5. Sonar Control Functions

a. Enable Sonar

Syntax: void **EnableSonar**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function enables the sonar group that contains *sonarNumber*, which causes all the sonars in that group to echo-range and write data to the data registers on the sonar control board.

b. Disable Sonar

Syntax: void **DisableSonar**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function removes the sonar *sonarNumber* from the *enabled_sonars* list. If sonar *sonarNumber* is the only enabled sonar from it's group, then the group is disabled, Otherwise, the group will continue echo ranging until all sonars in group are disabled.

c. Logging Sonar Data

Syntax: void **Sonarlog**(*Frequency*, *BufferSize*, *SonarNumber*, *LogType*)

Parameters: int *Frequency*, int *BufferSize*, int *SonarNumber*, int *LogType*;

Description:

This function prepares the tracing system to log sonar data. The *Frequency* specifies how many sonar cycles are skipped before data is logged. The *Buffersize* specifies how many bytes of storage to allocate to save the data. The *SonarNumber* indicates the sonar you wish to log data. The *LogType* specifies the type of data to be logged.

The type of logging data could be SONAR_RAW which logs the sonar range, SONAR_GLOBAL which logs the object's x, y coordinates and its range, SONAR_SEGMENT which logs segment data, and SONAR_ALL which indicates to log all three types of data.

d. Get Sonar Returns

Syntax: double **Sonar**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function returns the distance (in centimeters) sensed by the *sonarNumber*-th ultrasonic sensor. If no echo is received, an INFINITY (999999.0) is returned. If the distance is less than 10 cm, then a 0 is returned.

e. Get Global Sonar Returns

Syntax: POINT **Global**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function returns the data of type POINT which indicates the global x and y coordinates of the position of the last sonar return.

f. Get Segment

Syntax: Segment **GetSegment**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function returns the pointer to the oldest completed unread segment of the sonar passed in. If there is no completed unread segment NULL is returned.

g. Enable Linear Fitting

Syntax: void **EnableLinearFitting**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function enables the linear fitting algorithm to be active. The algorithm gather data points from sonar *sonarNumber* and form them into line segments.

h. Disable Linear Fitting

Syntax: void **DisableLinearFitting**(*sonarNumber*)

Parameters: int *sonarNumber*

Description:

This function causes sonar system to cease forming line segments for sonar *sonarNumber*.

X. SENSOR-BASED MOTION NAVIGATION

A. SENSOR-BASED MOTION CONTROL

The most typical way of using steering function is tracking a path specified by a configuration q . When the robot is tracking the path, Δk , $\Delta\theta$, and Δd can be computed by motion control subsystem to determine the motor control value and eventually be used to drive the wheels in a short interval (which is 10 ms in MML-11).

In addition to path tracking, there are some other flexible motion control methods which are helpful in local motion planning. This method requires integration of robot's sensors with motion control algorithm while not using path segment. Since motion control is basically based on the sensor returned information, we called it a *sensor-based motion control*.

The essential idea of this new method is based on the fact that obstacles present in the working environment and the sensors are able to detect those obstacles and to return their distances for processing. We assume that the obstacles are all of rectilinear polygons and they have walls and corner on their outer appearance. Therefore, it is possible for a robot to travel in the free space along obstacles' outer boundary and to keep certain constant safety clearance (Safety clearance concept is to be defined in Section B). We name this kind of robot's behavior as **wall-following**. Since keeping clearance from objects is important in wall-following motion, the robot will travel along a wall, follow a wall in other words, with clearance required when it is available or desired. But when a corner is eventually met in wall-following, the robot needs to change its orientation to keep following the object. During the robot changing its heading orientation, it is traveling along the corner, following the corner in other words, trying to keep the required clearance from the object so that it can continue to perform the same motion when a wall is available again. Since the walls and corners are interpreted as edges and vertices of polygons in geometry, we name the two different motions described above as *edge-following motion* and *vertex-following motion* respectively. The wall-following motion can be either *right-wall-following* or *left-wall-*

following depending upon right clearance or left clearance the robot is trying to keep from objects for navigation. For example, if the robot intends to keep its left clearance with objects, the wall on the robot's left will be used for navigation, and then the wall-following motion is called left-wall-following motion.

B. CLEARANCE DEFINITION

In wall-following Motion the robot will use sensors for navigation, thus safety clearance is an important factor. For a better understanding of how the sensors are used in motion planning, first we define some terminologies related to safety clearance. The **clearance w_0** is defined as the distance from the robot's outside edge of the wheels to the object. The distance data the sensor returns is raw data which indicates how far the object is from the sensor. With the sensor configuration in hand, the **clearance w_0** can be computed. The robot's **safety distance w_1** is defined as the summation of clearance w_0 and half of robot's width **HWidth**. Figure 10.1 illustrates the definitions of clearance elements.

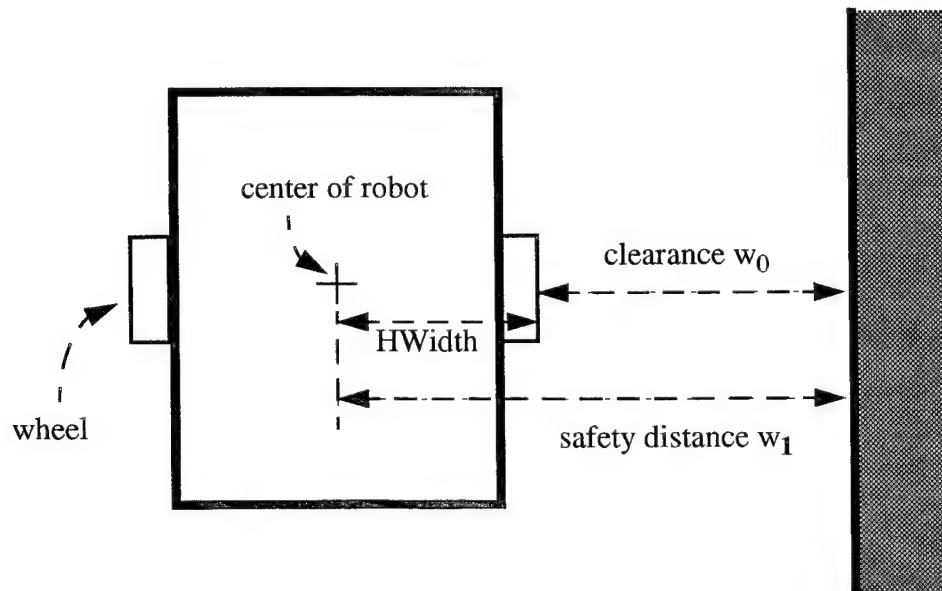


Figure 10.1: The Definitions of Robot's Clearance Elements

C. EDGE-FOLLOWING MOTION

While there is a wall on either side of the robot and robot is trying to keep itself away from the wall with a safety distance w_l , it is following an edge. We define this as the robot is in *edge-following mode*. We apply the steering function in Eq 3.1 to control the robot while the robot is in edge-following mode. Let the robot's current configuration be $q = (p, \theta, k)$. The variables Δk , $\Delta \theta$, and Δd in steering function can be computed based on some assumptions as followings. The desired curvature of the wall is zero because we assume the wall is flat like a line. The desired orientation of the wall, θ_d , can be computed before the robot is following the wall. This desired orientation is orthogonal because we assume the environment consists of rectilinear polygons. The distance between the robot and the wall, d_w , can be obtained by using sensors. Therefore the variables for the steering function are defined as:

$$\Delta k = k;$$

$$\Delta \theta = \theta - \theta_d;$$

$$\Delta d = w_l - d_w;$$

Figure 10.2 shows the robot in edge-following mode in left-wall-following type

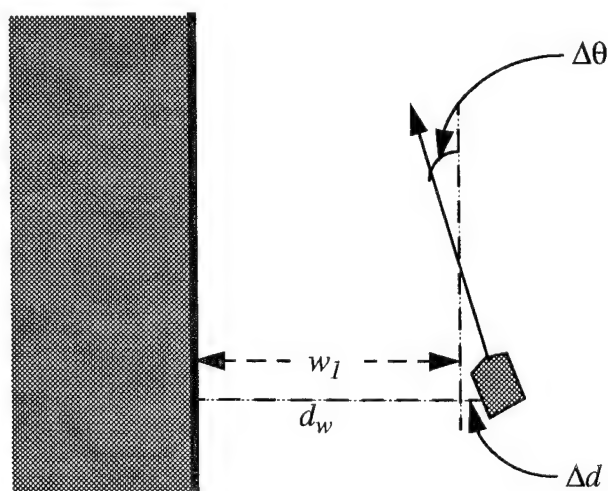


Figure 10.2: The Edge-following Mode in Left-wall-following

D. VERTEX-FOLLOWING MOTION

In a normal polygon (convex polygon), when the robot is coming to the end of an edge, the sensors can detect a convex corner which is interpreted as a convex vertex. In order to keep safety clearance from the object, the robot need to turn around the vertex with a circle motion taking the vertex as its center (that was obtained by sensor) and safety clearance w_l as its radius. We define the second mode, named *vertex-following mode*, in the sensor-based motion control for this situation. Let's take left-wall-following as an example. The curvature of the circle then is calculated as $1/r$, where r is the circle's radius. Let the robot's current configuration be $q = (p, \theta, k)$ and θ_d be the desired orientation on the circle. The distance between robot's current position and center of the circle, d_c , can be easily computed. Therefore, the steering function variables are computed as below:

$$\Delta k = k - 1/r;$$

$$\Delta \theta = \theta - \theta_d;$$

$$\Delta d = r - d_c;$$

Figure 10.3 illustrates the vertex-following in left-wall-following.

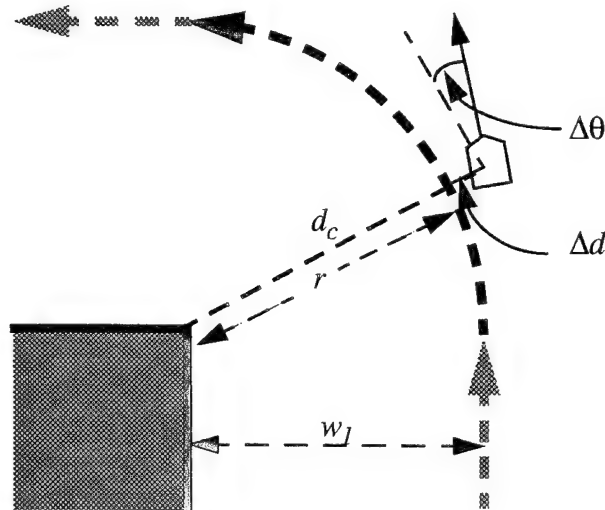


Figure 10.3: The Vertex-following Mode in Left-wall-following

For an inverted polygon or a general polygon, the corner can be either convex or concave one. The motion of following a convex corner are described as above. For a concave corner, the same idea of vertex-following stated above will be applied to the concave vertex situation except the calculation of the center (the vertex) of the circle. Figure 10.4 illustrates the computation of the center of circle vertex-following on concave corner in left-wall-following.

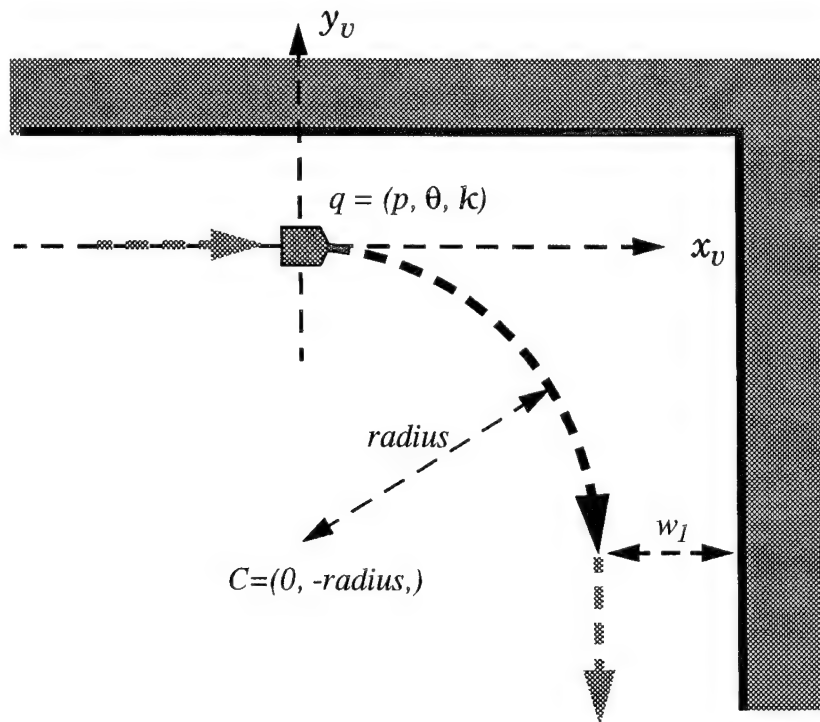


Figure 10.4: The Vertex-Following on Concave Corner

E. WALL-FOLLOWING MOTION CONTROL

This section describes how we control the robot's motion in wall-following motion planning. We define four states for controlling robot in accordance with the modes defined in wall-following motion as follows:

- **EDGE,**

- **EDGE0**,
- **CONVEX**, and
- **CONCAVE**.

When the robot is in the edge-following mode, the state is defined as **EDGE** state. The robot will adjust itself to follow the wall with given safety distance w_I by computing steering function variables as described previously. When the robot is in the vertex-following mode and the corner is found as a convex one, we define the robot's motion is in **CONVEX** state. When the robot is in the vertex-following mode and the corner is a concave one, we define it's in **CONCAVE** state. In both **CONVEX** and **CONCAVE** states, the robot will take the computed center as a vertex and turn around the vertex with a circle motion. Because of the limitation of sensors, it is difficult to compute the exact position of circle's center for the motion to follow a convex corner. Therefore, an intermediate state **EDGE0** is created to prevent unexpected distance returned from sensors being used. The intermediate state **EDGE0** is a transition state between **CONVEX** state and **EDGE** state. The transitioning from **CONVEX** to **EDGE0** is controlled by checking the angle the robot turned in the circle motion. When the robot has turned desired angle (normally it is 90 degree), the state changes from **CONVEX** to **EDGE0**. While the robot is in **EDGE0** state, it follows a desired path, which is computed at the beginning of **CONVEX** state, until it passes the point where the robot can start the edge-following. Then the robot changes the state from **EDGE0** to **EDGE**.

At the beginning of Wall-Following motion, the robot is assumed in **EDGE** state. While the robot is following the edge, the state can be changed according to the environment. If the corner is not detected, it stays in **EDGE** state. If a convex corner is identified, the robot switches its state from **EDGE** to **CONVEX**. Similarly, if a concave corner is found, the robot switches its state from **EDGE** to **CONCAVE**. While the robot is in **CONVEX** or **CONCAVE** states, the sensors are not used. Before the turning of desired angle is finished, the robot stays in its original state. At the end of **CONVEX** states, the robot switches the state to **EDGE0**. Then it switches to the **EDGE** state according to the

position checking. Figure 10.5 illustrates the states change in wall-following in a convex corner. The state change from CONCAVE is directly to EDGE when the vertex-following is finished. Figure 10.6 illustrates the state change in a concave corner. Figure 10.7 shows the transition of robot's states in wall-following motion.

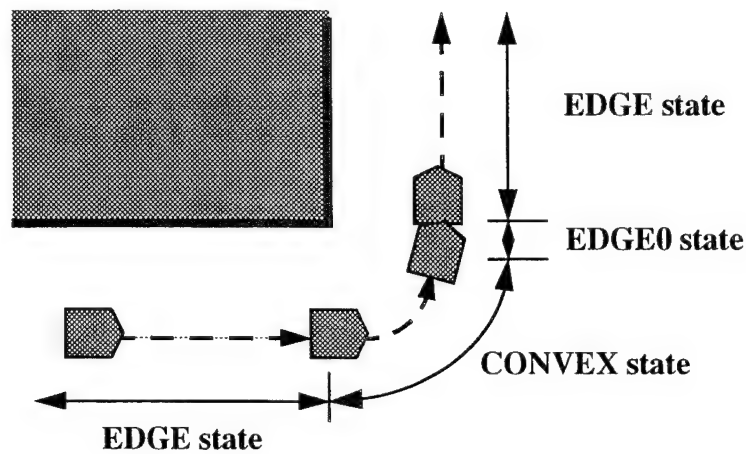


Figure 10.5: The State Change in Wall-following Motion I

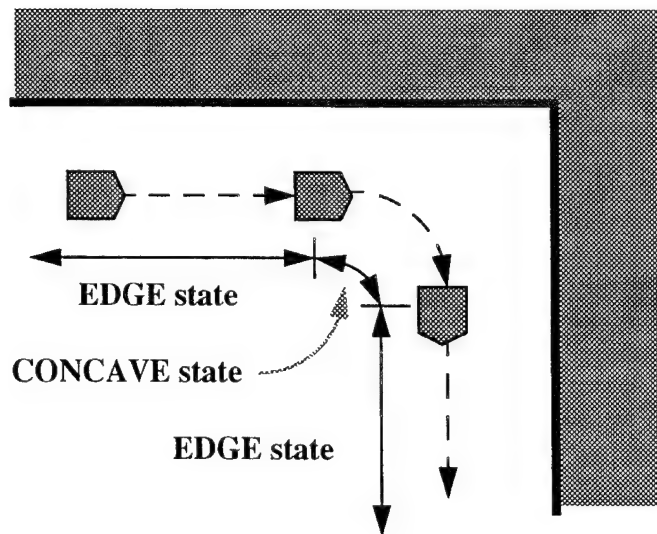


Figure 10.6: The State Change in Wall-following Motion II

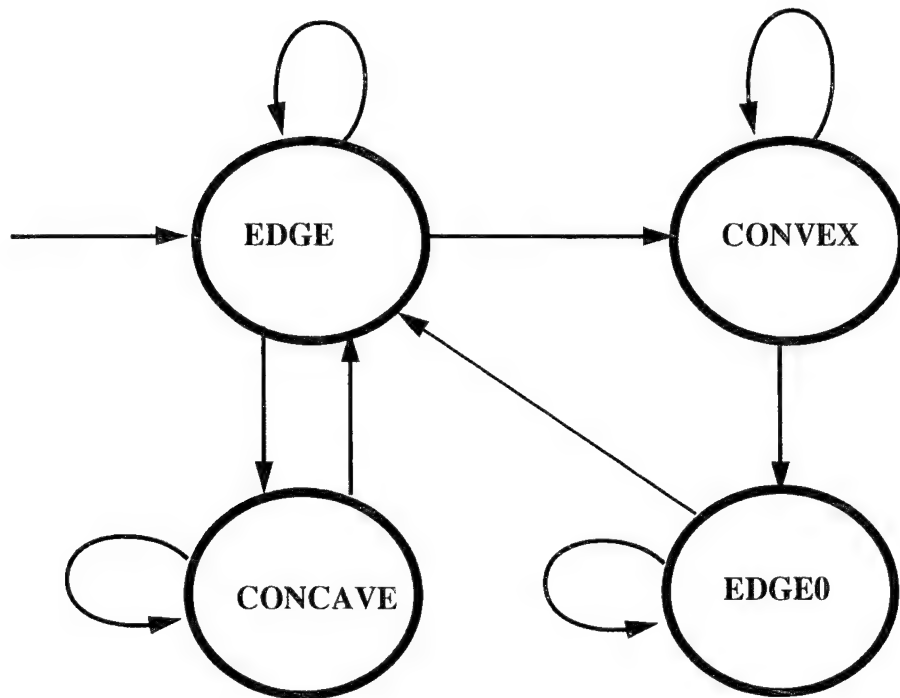


Figure 10.7: The State Transition of the Wall-following Motion

F. ALGORITHM

The algorithm for the wall-following motion is as following:

Algorithm FollowRule (va, vc)

Input: va : the actual velocity. vc : the commanded velocity.

Output: Commanded linear and rotational speeds

- (1) get the desired clearance $w0$
- (2) case **EDGE** :
- (3) get sensor distance $dist$
- (4) if convex corner not found then
- (5) if concave corner not found then
- (6) get vertex for center of circle motion
- (7) get desired orthogonal orientation
- (8) compute Δd of edge-following
- (9) else
- (10) compute circle radius
- (11) compute circle curvature
- (12) compute center of circle

```

(13)    compute desired orthogonal orientation
(14)    compute robot's desired orientation
(15)    compute desired curvature
(16)    compute  $\Delta d$  of vertex-following
(17)    switch state to CONCAVE
(18)    case EDGE0 :
(19)      if not finish turning then
(20)        compute  $\Delta d$  of the desired path following
(21)      else
(22)        compute  $\Delta d$  of edge-following
(23)        switch state to EDGE
(24)    case CONCAVE :
(25)      if not finish turning then
(26)        compute robot's desired orientation
(27)        compute  $\Delta d$  of vertex-following
(28)      else
(29)        compute robot's desired orientation
(30)        compute desired curvature
(31)        compute  $\Delta d$  of edge-following
(32)        switch state to EDGE
(33)    case CONVEX :
(34)      if not finish turning then
(35)        compute robot's desired orientation
(36)        compute  $\Delta d$  of vertex-following
(37)      else
(38)        compute robot's desired orientation
(39)        compute desired curvature
(40)        compute  $\Delta d$  of edge-following
(41)        switch state to EDGE0
(42)    compute  $\Delta k$  and  $\Delta \theta$ 
(43)    compute  $\Delta k / \Delta s$  (using steering function)
(44)    compute commanded linear and rotational speeds
(45)    return commanded linear and rotational speeds

```

G. IMPLEMENTATION

The robot's motion is controlled by a motion control system. In Yamabico-11 this module is invoked every 10 milliseconds. To implement wall-following motion, we need to create new motion rules other than the regular motion rules which are used for path tracking motion control. These rules are named **LeftFollowRule** and **RightFollowRule**.

1. Simulation

The use of simulation to verify the theory is important in scientific research. As we know the desired curvature of edge-following is zero since we are assuming the wall is a

flat wall which will be a straight line if it is projected on the two dimensional plan. But the desired curvature of vertex-following will be the curvature of desired circle which the robot is supposed to follow. In order to maintain a smooth motion, the robot's curvature is not allowed to change abruptly. We were not sure whether the robot's curvature would be changing smoothly or not when the following mode changed. Instead of implementing it on real robot directly, we first simulate the mode change. Figure 10.8 illustrates the trajectory of wall-following motion simulation. Because we can not simulate sonar's operation, we assume the sonar always returns the safety distance. In other words, the Δd is always zero. Therefore, the simulated trajectory is actually a reference path for the real time robot's motion. From this simulation results, we know that the robot's curvature will change smoothly while it switches from one following mode to another.

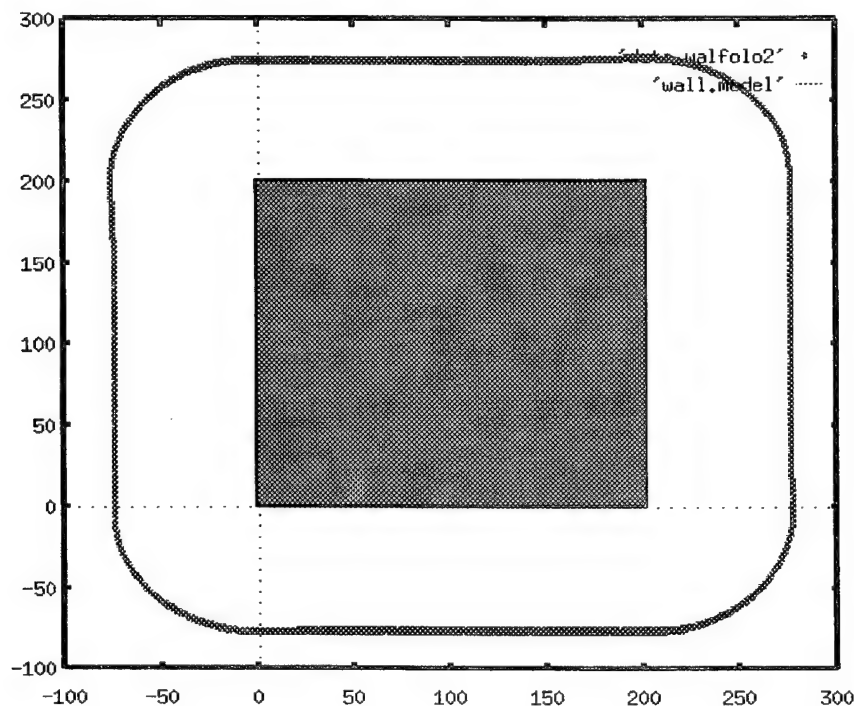


Figure 10.8: The Trajectory of Wall-following Simulation

2. Experimental Results of Real Time System

The implementation of wall-following motion will naturally require the utilization of sensors. In Yamabico-11, the available sensors are 12 sonars which are installed around the robot with 30 degrees apart from one another in their orientations. Figure 6.3 illustrates the configurations of the sonars in Yamabico-11.

To follow a wall, the left and right sonars, # 5 and # 7, will be used to detect the distance of the wall on right and left sides respectively, so that the motion rules can steer the robot to follow a wall keeping the specified safety distance from the wall. The sonar # 0 in front of robot is mainly used to check the wall in front.

A lot of experiments have been done using real time robot after simulation succeeded. Figure 10.9 illustrates the trajectory of the real time robot Yamabico-11 traveling around a box with right-wall-following motion. This proves that the robot is able to navigate itself by using its sensors in an unknown environment.

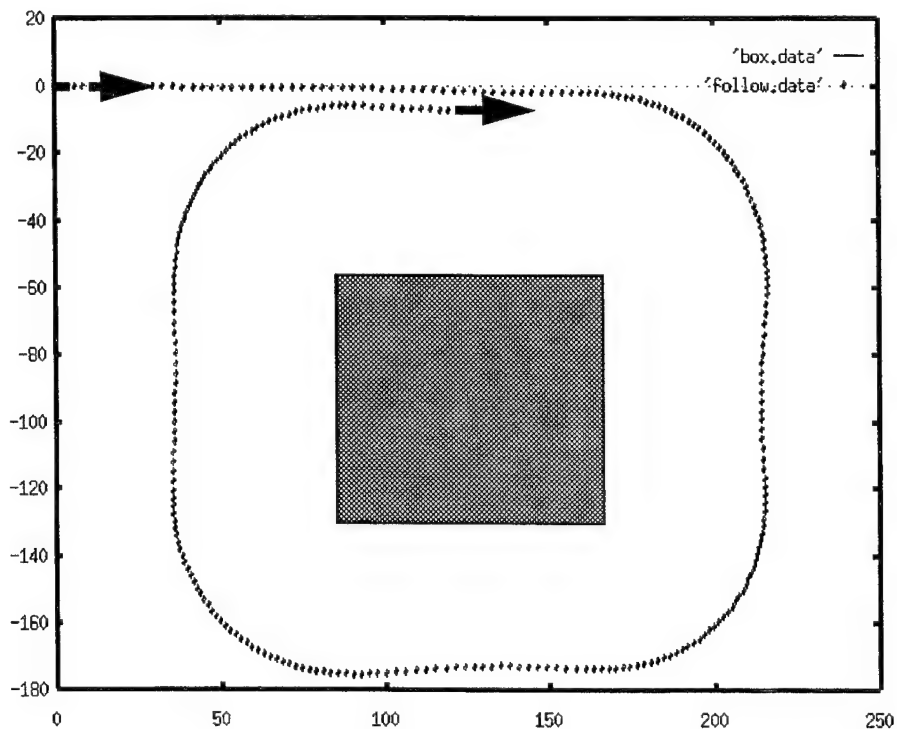


Figure 10.9: The Trajectory of Real Time Wall-following Motion

XI. IMPLEMENTATION

A. OVERVIEW

This chapter describes how to implement the local motion planning. The implementation of global motion planning was described in [7]. In the chapter, the data structures for end-portion motion planning and mid-portion planning are addressed first. Then the algorithms of implementation are discussed. At last, the experimental results conducted by Yamabico-11 using MML-11 software system will be presented.

B. DATA STRUCTURES

This section describes the data structures for implement local motion planning. Since end-portion motion planning is more complicated, it requires a data structure which is different from the data structure used in mid-portion motion planning to hold the motion instructions. We will describe the data structure for implementing end-portion motion planning first. After that, the data structure for implementing mid-portion motion planning will be discussed.

1. Data Structure for End-portion Motion Planning

In the subsection we discuss the main data structure to hold motion instruction of the results of end-portion motion planning. The intermediate data structures required by planning operations are straightforward as described in Chapter V. In end-portion motion planning, both forward and backing up motion may be required to accomplish the task. Thus, the data structure needs to reflect this requirement in motion instruction. In order to make motion control (execution) as simple as possible, for any single motion, the data structure required includes:

- **Reference path:** The reference path stores the whole path information which will be followed. A path is normally represented by configurations which include position, orientation and curvature. Since the reference path could be a straight line or a curve generated by simulator, it may consist one or more than

one configuration. Thus, a dynamic structure is needed for this data element to store finite number of configurations. A linked list will be used to construct the reference path. (see Figure 11.1).

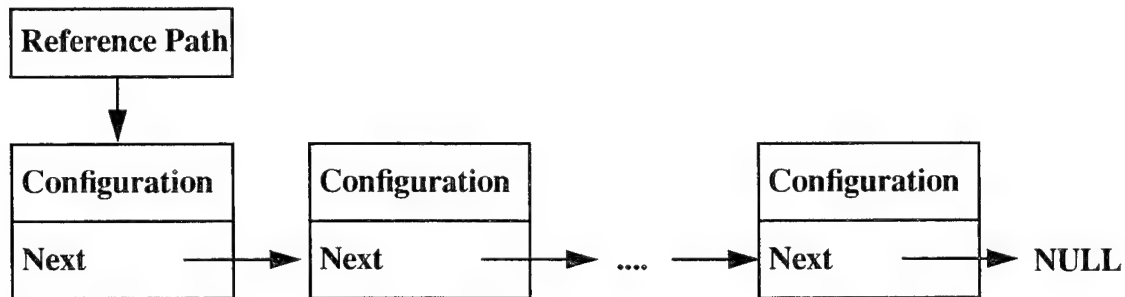


Figure 11.1: Reference Path Data Structure

- **Sigma:** The sigma is an important information for conducting path tracking. It will be used to decide the smoothness of tracking trajectory while this motion instruction is taken. The type of this element is double.
- **Motion type:** This element indicates whether the motion is forward motion or backing up motion.
- **End configuration:** This is a type of configuration which indicates the last configuration the robot needs to transition to next motion instruction.
- **Stop:** This element indicates whether the motion need to stop or not. Normally, a backing up motion needs to stop at the end configuration as indicated. A forward motion may need to stop in the case of a backing up motion following.
- **Next:** This is a pointer pointing to next motion instruction.

As aforementioned, end-portion motion could be a combination of many forward and backing up motions. However, the number of different motions is dynamic. Therefore, the data structure should be designed to satisfy this need. Figure 11.2 illustrates the entire data structure for end-portion motion planning.

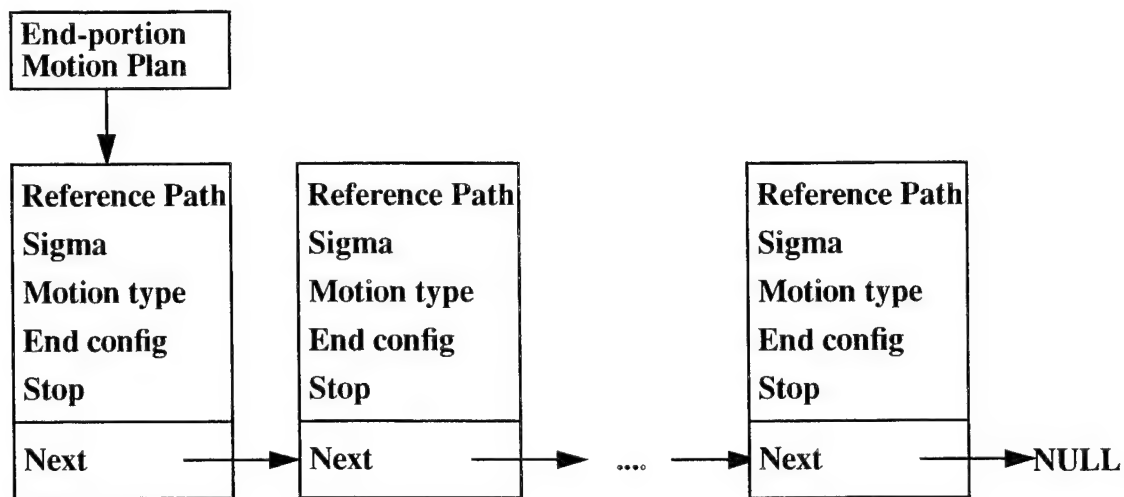


Figure 11.2: End-portion Motion Data Structure

2. Data Structure for Mid-portion Motion Planning

Although there are many different kind of combined motions in the results of mid-portion motion planning, we can design a simple data structure to contain all kind of motion instructions. The basic information needed for the motion in a region includes:

- Region ID,
- Entrance configuration,
- Reference path,
- Sigma,
- Initial tracking configuration,
- Exit border ID and its corresponding,
- Exit configuration.

For instance, the simplest motion can be planned to track a reference line of exit configuration of a region. The example can be found in Figure 4.13. To specify this motion, we can store the exit configuration in Reference path and indicate where to start the tracking in Initial tracking configuration and store the proper smoothness in Sigma. Then the robot can conduct the motion as shown in the figure. Is it possible to use such a simple

data structure to specify a more complicated motion as the two perpendicular line tracking in Figure 4.16 (b)? The answer is yes. As long as the information of reverse path and the initial configuration, where the robot needs to start its path tracking, are stored, the robot is able to perform the desired motion. This is because the reference path can be designed as a linked list which stores a sequence of configurations that specify the reverse path and the first configuration of the reversed path specifies a line which is exactly the reference line of the first perpendicular line tracking. Therefore, in implementing mid-portion motion planning, the data structure required is designed as a linked list as shown in Figure 11.3. In the Figure 11.3, each node of the linked list holds the motion instruction for one region of the mid-portion of global path.

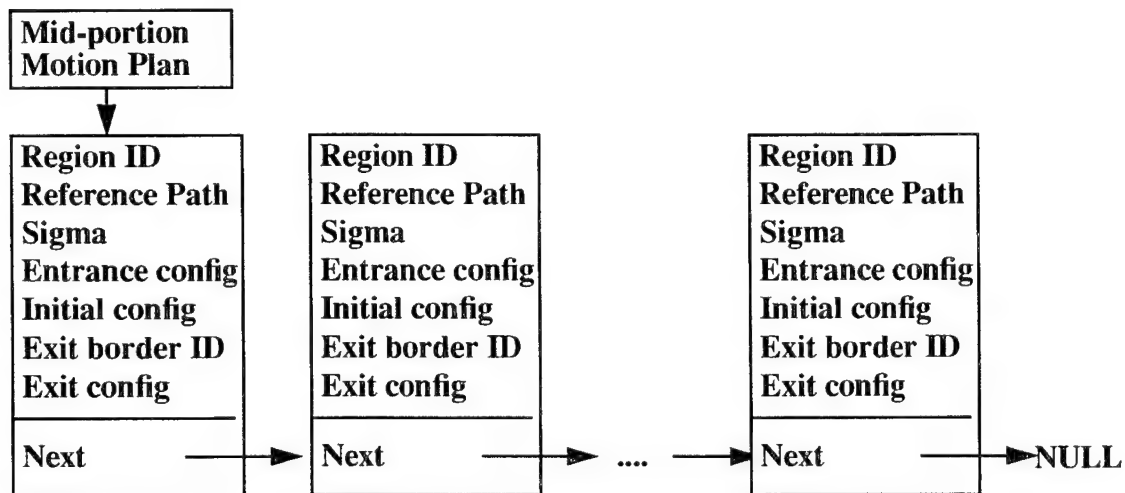


Figure 11.3: Mid-portion Motion Data Structure

The **Region ID** and **Exit border ID** are the type of integer which indicate the current region and its exit border. The **Reference path** of the structure is another linked list as shown in Figure 11.1. The **Sigma** is a type of double. The **Entrance config**, **Initial config** and **Exit config** are all type of configuration.

C. LOCAL MOTION PLANNING ALGORITHMS

An overall algorithm for local motion planning are presented in Chapter III (p. 42). The mid-portion motion planning and end-portion motion planning algorithms are described in detailed in Chapter IV and V respectively. Keep in mind that for symmetric motion planning, the final motion planning is conducted in reverse manner. All related configurations must be reversed while planning.

D. EXPERIMENT RESULTS

Most of motion planning algorithms described in this dissertation have been implemented in MML-11 and tested on experimental robot Yamabico-11. The results shows that the algorithms are practical to the robot motion planning and motion control. Figure 11.4 through 11.8, show the trajectories of motion executions with various given start and goal configurations on the model of the fifth floor in Spanagel Hall, NPS.

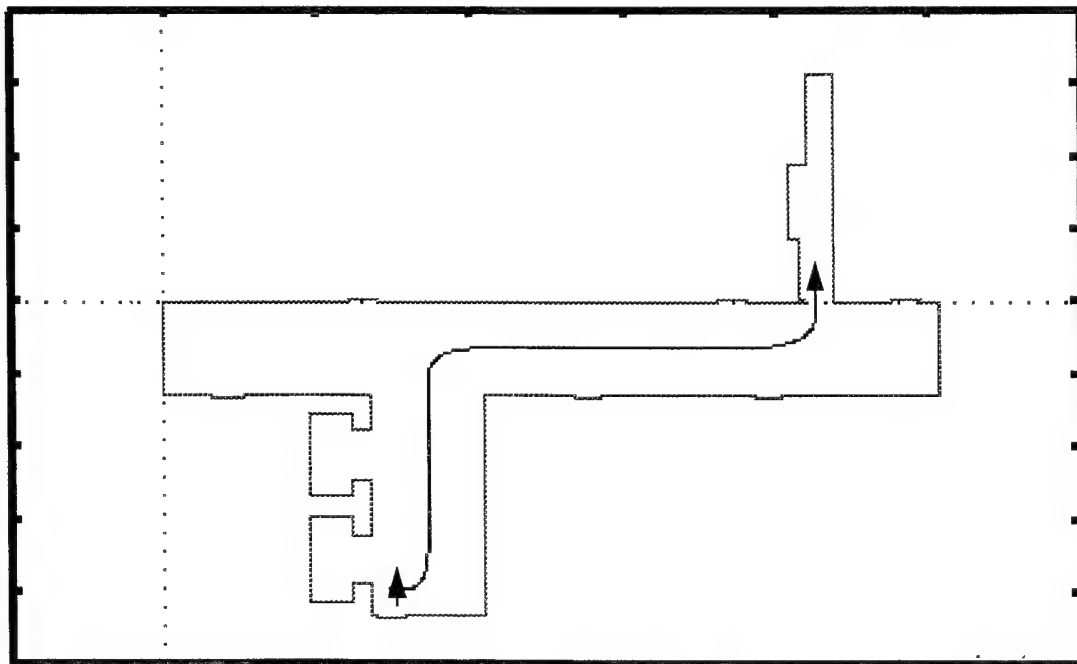


Figure 11.4: Yamabico-11 Motion Planning and Execution Results #1

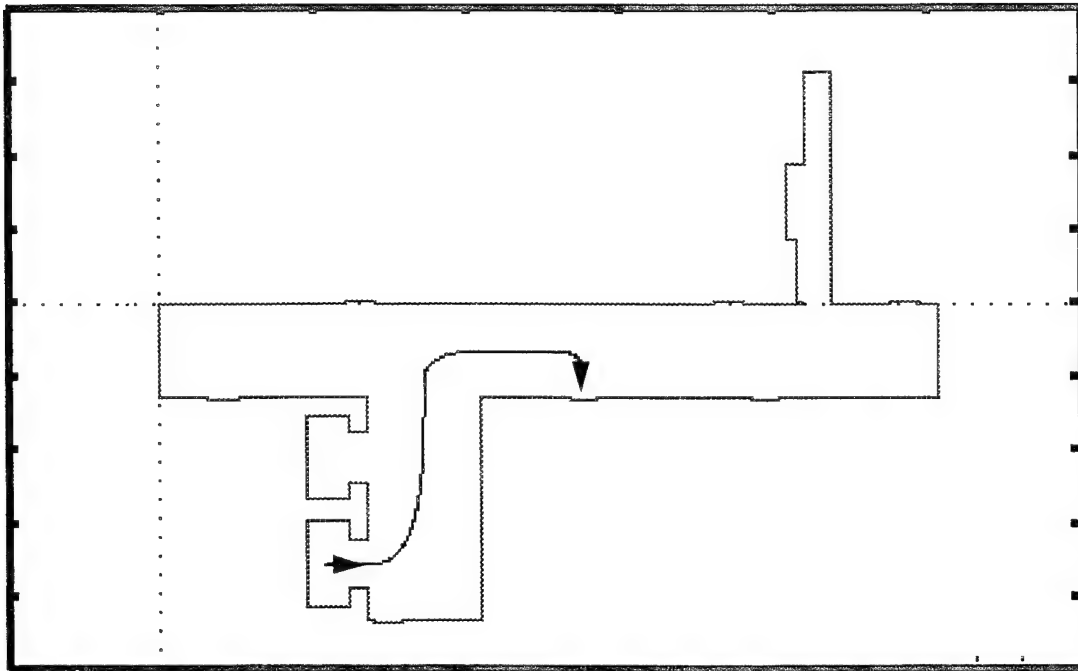


Figure 11.5: Yamabico-11 Motion Planning and Execution Results #2

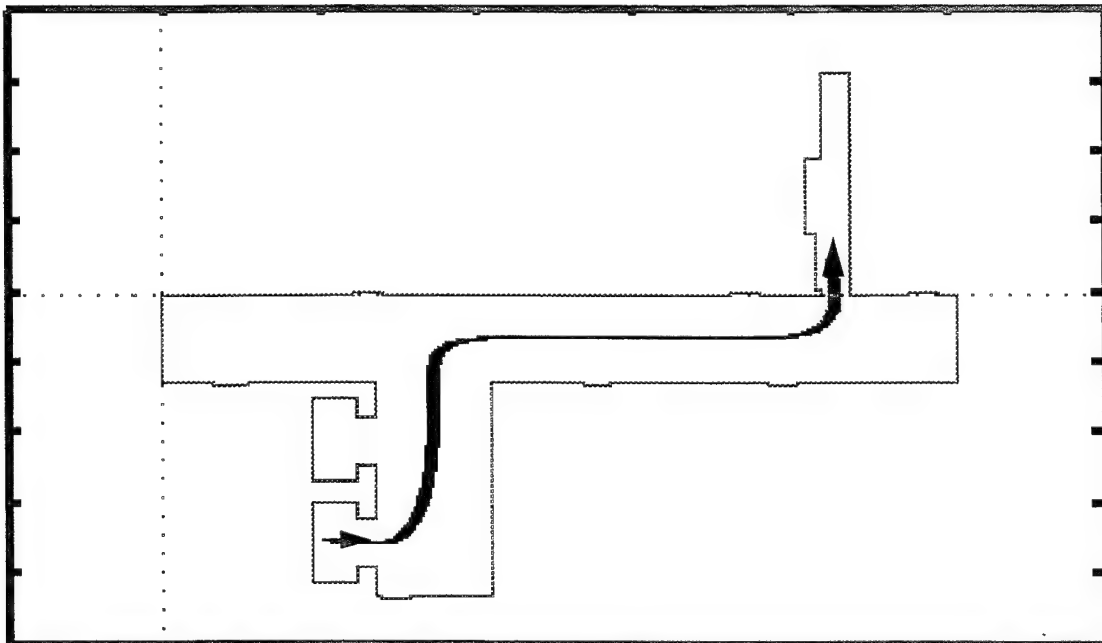


Figure 11.6: Yamabico-11 Motion Planning and Execution Results #3

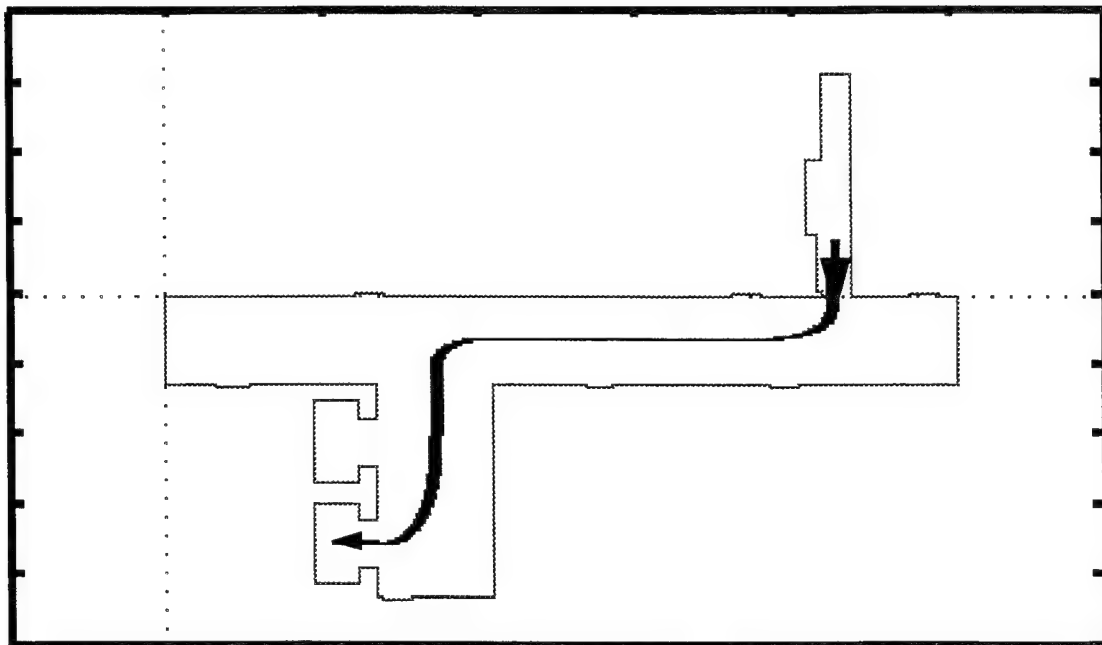


Figure 11.7: Yamabico-11 Motion Planning and Execution Results #4

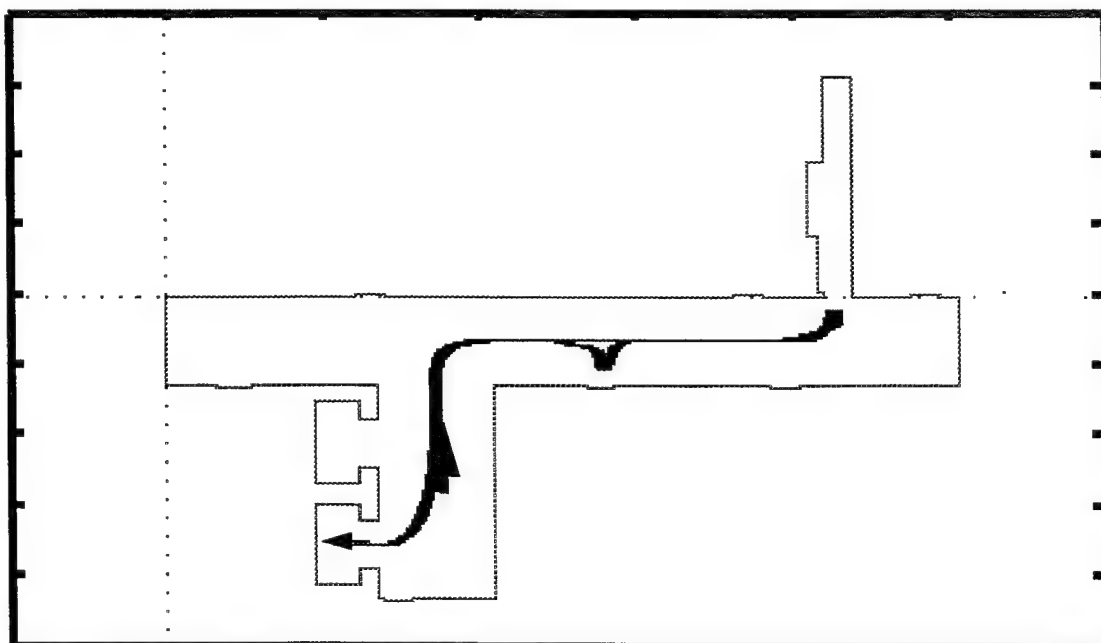


Figure 11.8: Yamabico-11 Motion Planning and Execution Results #5

XII. CONCLUSIONS

This dissertation addressed safe motion planning using layered planning approach which divides the planning task into global path planning and local motion planning. Among the researches in the subject of motion planning for mobile robots, the contribution of this dissertation provides a practical solution to safe motion planning problem which is a great step in promoting motion planning in the real world. The local motion planning is accomplished by planning motion in end-portion and mid-portion of the global path separately. Under the safety consideration, the smoothest motion is achieved by dynamically computing proper smoothness variable for reference path generation and path tracking. This dissertation analyzed various region situations and summarized the motion planning of any single region into six rules for mid-portion motion planning. These rules simplify the local motion planning and enhance planning efficiency.

There are some other contributions including: (i) design of real-time robot operating system which makes robot system control and other robot sensing devices control work on the interrupt-driven basis. (ii) incorporating forerunner simulation into real-time transition point calculation to make robot motion control more flexible. (iii) development of a standard high level robot language for motion planning and robot system control.

In addition, the steering function was studied intensively through many simulations and experiments. The characteristics of limitation of the powerful motion planning and control tool are made clear.

Those research results were implemented in a software system, MML-11 and tested on Yamabico-11. The experiment results show that the algorithms are successful in robot motion planning and motion control.

APPENDIX A. FURTHER UNDERSTANDING OF STEERING FUNCTION

The steering function as Eq A.1 is a powerful tool for a mobile robot vehicle performing smooth path tracking.

$$\frac{dk}{ds} = -(A\Delta k + B\Delta\theta + C\Delta d) \quad (\text{Eq A.1})$$

In Eq A.1, A, B, and C are positive constants which are related to the smoothness of robot's motion [19]. The meanings of these variables, Δk , $\Delta\theta$, and Δd , are as follows: $\Delta k = k - k_d$, where k is vehicle's current curvature and k_d the desired curvature. $\Delta\theta = \theta - \theta_d$, where θ is vehicle's current orientation and θ_d the desired orientation. Δd is the vehicle's position error. How the steering function accomplishes a path tracking is fully described in [19], [21] and [7]. To better use the steering function in motion planning, we need to understand its characteristics deeply. First of all, we investigate the limitation of smoothness which will be applied to a vehicle in path tracking using the steering function. Then we look for a possible alternative way of using steering function.

A. LIMITATION OF SMOOTHNESS ON LINE TRACKING

The path of the path tracking in this appendix refer to a straight line. Thus we rephrase the term path tracking as *line tracking*. A line tracking motion starts from a configuration $q_s = (p_s, \theta_s, k_s)$. The goal of the line tracking is the line called reference line, specified by a configuration $q_r = (p_r, \theta_r, k_r)$. Since the line is a straight line, it has a constant curvature $k_r = 0$. The constants A, B, and C in the steering function are determined by the smoothness σ as follows:

$$k = 1 / \sigma \quad (\text{Eq A.2})$$

$$A = 3 k \quad (\text{Eq A.3})$$

$$B = 3 k^2 \quad (\text{Eq A.4})$$

$$C = k^3 \quad (\text{Eq A.5})$$

While the vehicle is tracking a line, its curvature is kept updated by steering function in each motion cycle [7] until the vehicle's trajectory converges to the reference line. Thus the smoothness σ in turns determines the sharpness of the tracking trajectory. Figure A.1 illustrates the change of sharpness of trajectories when different smoothnesses are applied in the simulation. In fact, we see that the smaller the value of smoothness is, the sharper the trajectory will be. And we also observed that the larger the smoothness is, the longer the vehicle travels to make its trajectory converge to the reference path.

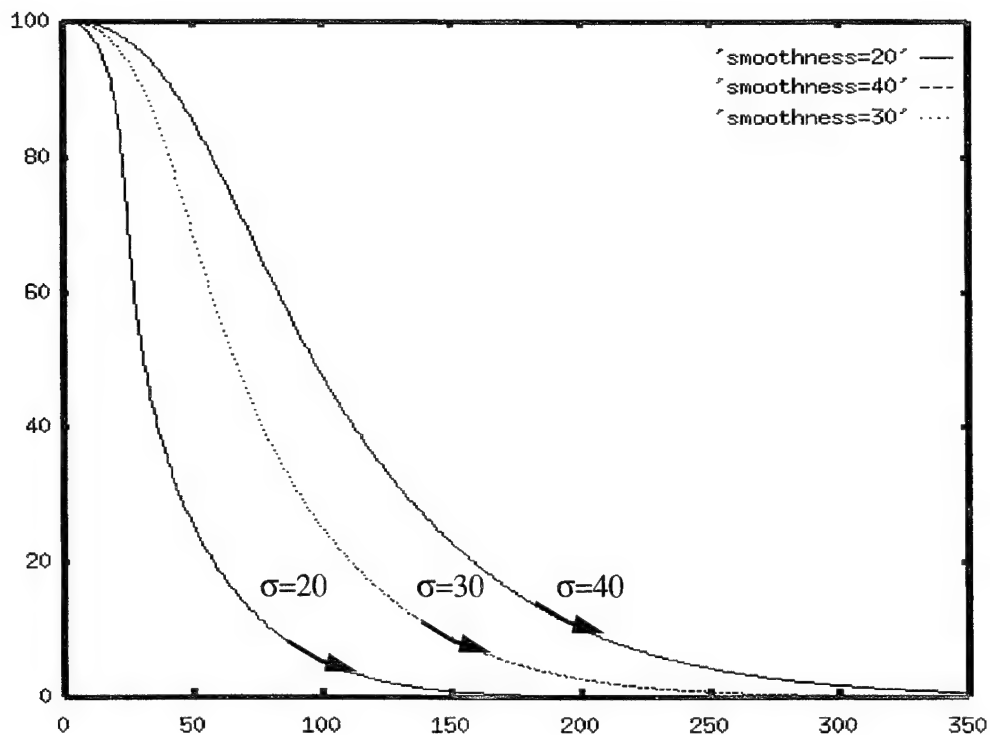


Figure A.1: The Sharpness of Tracking Trajectory Affected by Different Smoothness

While we know how the smoothness affects the tracking trajectory, we have to be aware of the limitation of smoothness that can be applied to a line tracking. It is important to know that not all arbitrary smoothness can be used in all cases of line trackings. As a matter of fact, there is no upper limit for the smoothness in a line tracking, even though an

oscillatory trajectory might occur if the applied smoothness is too large. However, we notice that a lower limit of smoothness does exist in using the steering function to track a line.

By observing the simulation results of using steering function in line tracking, we found that the lower limit of smoothness is tightly related to the initial Δd which is the closest distance from the initial configuration to the reference line. Figure A.2 illustrates the results of simulations using some critical smoothnesses in line tracking. These simulation were set by tracking positively oriented X-axis of the global frame from a configuration with initial $\Delta\theta = 0$ and $\Delta d = 100$. Because the initial $\Delta\theta = 0$ means the orientation of the initial configuration and the reference line are parallel, this type of line tracking is called *parallel line tracking*. From these parallel line tracking simulation, we found that the smallest smoothness which makes the tracking trajectory converge to the reference line is

$$\sigma = 0.096 * \Delta d_{\text{init}} \quad (\text{Eq A.6})$$

where Δd_{init} represents the initial Δd . If a smoothness σ less than $0.096 * \Delta d_{\text{init}}$ is applied, the tracking trajectory never converges to the reference line, instead, it converges to a line which is parallel to the reference line. This case is shown as the trajectory π_1 in Figure A.2 where the smoothness is $\sigma = 0.095 * \Delta d_{\text{init}}$.

The non-converging (to the reference line) situation happens when the tracking trajectory reaches the point where the variables of the steering function in Eq A.1 are as follows:

$$\Delta k = 0; \quad (\text{Eq A.7})$$

$$\Delta\theta = 2n\pi \text{ for an integer } n; \quad (\text{Eq A.8})$$

$$\Delta d \neq 0;$$

$$\text{and } \frac{dk}{ds} = 0;$$

Thus, we have

$$B\Delta\theta + C\Delta d = 0 \quad (\text{Eq A.9})$$

Then by substitute Eq A.9 with Eq A.4, Eq A.5 and Eq A.2, we have

$$\Delta d = -\frac{B\Delta\theta}{C} = -\frac{3k^2 \times 2n\pi}{k^3} = -6n\sigma\pi; \quad (\text{Eq A.10})$$

As the point on the tracking trajectory matches Eq A.7, Eq A.8 and Eq A.10, the steering function can no longer steer the vehicle to the reference line. Therefore we use the steering function for line tracking, the proper smoothness must be carefully chosen to avoid the non-convergence from happening.

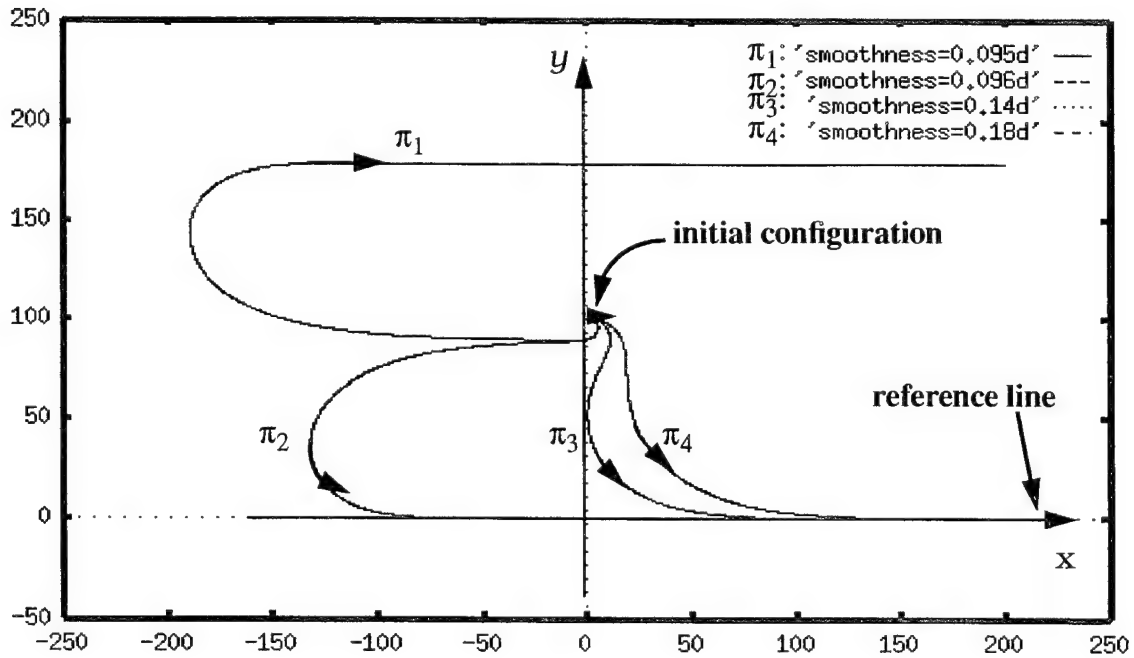


Figure A.2: The Trajectories of Line Tracking with Various Smoothness

In addition to the worst case described above, π_2 and π_3 in Figure A.2 demonstrate other undesirable line tracking trajectories. Although they converge to the reference line eventually, the trajectories travel backward at first half. On the contrary, we consider π_4 as a desirable tracking trajectory because it always travels in the direction of the reference line heading. From these simulations, we conclude that for parallel line tracking the minimum desirable smoothness (which makes a desirable converging trajectory) is:

$$\sigma = 0.18 * \Delta d_{init} \quad (\text{Eq A.11})$$

The Figure A.2 demonstrates only the simulation results of a parallel line tracking where we set initial $\Delta\theta = 0$. When the initial $\Delta\theta$ is not zero, this variable is also found involved in the determination of smoothness. Fortunately, since in any cases both initial Δd and $\Delta\theta$ are known, by using well-designed simulation, the minimum smoothness can be computed. Table A.1 and Table A.2 list some simulation data showing the relationship between initial Δd and its minimum desirable smoothness σ . From the tables we found that for all different initial orientations, the minimum smoothness is less than $0.22 * \Delta d_{init}$. We will take this Max-Min smoothness in some cases in Initial-portion Motion Planning.

Table A.1: The Relationship among Distances and Smoothness in Line Tracking with Negative $\Delta\theta$ (in degree), and Corresponding Minimum desirable Smoothness σ

<i>Initial $\Delta\theta$</i>	<i>d_{init}</i>	σ	<i>L</i>	σ/d_{init}	<i>L / σ</i>
0	100.0	18.0	177.3	0.18	9.85
15	100.0	19.0	191.6	0.19	10.09
30	100.0	19.0	191.7	0.19	10.09
45	100.0	20.0	204.6	0.20	10.23
60	100.0	20.0	203.2	0.20	10.16
75	100.0	21.0	214.4	0.21	10.21
90	100.0	21.0	211.9	0.21	10.09
105	100.0	22.0	222.3	0.22	10.10
120	100.0	22.0	219.1	0.22	9.96
135	100.0	22.0	216.3	0.22	9.83
150	100.0	22.0	213.8	0.22	9.72
165	100.0	22.0	211.6	0.22	9.62
180	100.0	22.0	209.7	0.22	9.53

Table A.2: The Relationship among Distances and Smoothness in Line Tracking with Negative $\Delta\theta$ (in degree), and Corresponding Minimum desirable Smoothness σ

<i>Initial $\Delta\theta$</i>	<i>d_{init}</i>	σ	L	σ/d_{init}	L / σ
-15	100.0	18.0	175.7	0.18	9.76
-30	100.0	18.0	173.1	0.18	9.62
-45	100.0	18.0	169.9	0.18	9.44
-60	100.0	18.0	165.8	0.18	9.21
-75	100.0	18.0	161.4	0.18	8.97
-90	100.0	22.0	200.0	0.22	9.09
-105	100.0	19.0	163.4	0.19	8.60
-120	100.0	16.0	123.9	0.16	7.75
-135	100.0	15.0	106.5	0.15	7.10
-150	100.0	13.0	70.2	0.13	5.40
-165	100.0	12.0	41.3	0.12	3.44

B. TWO-WAY LINE TRACKING

Although the steering function is designed under the assumption of $-\pi/2 \leq \Delta\theta \leq \pi/2$, it is applicable to all orientation differences. For a given initial configuration and a reference line with a proper smoothness, the line tracking can be performed smoothly and its trajectory is unique. Figure A.3 shows the trajectories of tracking a reference line (positively oriented X-axis) from the initial configurations which are different in their orientations only. The trajectory π_1 starts the line tracking from the initial configuration $((0, 100), -180, 0)$ and trajectory π_8 starts from $((0, 100), 135, 0)$. The orientations in those initial configurations vary from $-\pi$ to $3\pi/4$. In steering function point of view, the orientation differences are $-\pi \leq \Delta\theta \leq 3\pi/4$. It looks nature to have the trajectories as Figure A.3 for this line tracking example. However, in some cases, we may expect the vehicle to

travel a trajectory different from the one shown in Figure A.3 (starting at the same initial configuration and tracking the same reference line). For instance, the tracking trajectory π_8 in Figure A.3 starts with a clockwise motion at the beginning. If there is no way to avoid a possible collision by traveling this clockwise trajectory, we may try a possible counterclockwise tracking trajectory. The Figure A.4 illustrate the idea of clockwise and counterclockwise tracking trajectory. In Figure A.4, clockwise trajectory π_{cw} is exactly the same as π_8 in Figure A.3. The method we use to tracking a line either clockwise or counterclockwise ins called *Two-way Line Tracking*.

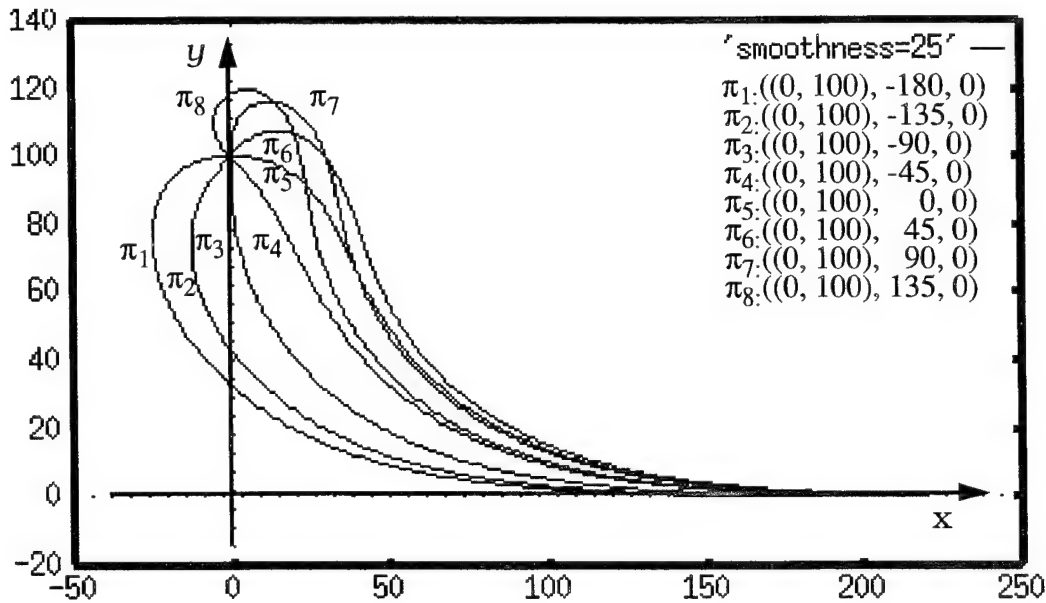


Figure A.3: The Trajectories of Line Tracking from Different Orientation

The steering function of Eq A.1 is capable for the Two-way Line Tracking. We notice that the line tracking trajectories in Figure A.3 have their initial orientations expressed as $-\pi \leq \theta < \pi$. Therefore, when tracking a line from those configurations, we have the orientation difference also as $-\pi \leq \Delta\theta < \pi$. As we know, any orientation difference can be normalized to this orientation interval. We considered a line tracking with the normalized orientation difference as a *normal line tracking*. Besides the normal one, if the

initial orientation difference is expressed by the following rule,

if $\Delta\theta < 0$, $\Delta\theta = \Delta\theta + 2\pi$.

else $\Delta\theta = \Delta\theta - 2\pi$.

the counterclockwise tracking trajectory as π_{ccw} in Figure A.4 is produced. We consider the line tracking with this orientation difference as an *alternative line tracking*. The Figure A.5, A.6 and A.7 demonstrate some more examples of Two way Line Trackings.

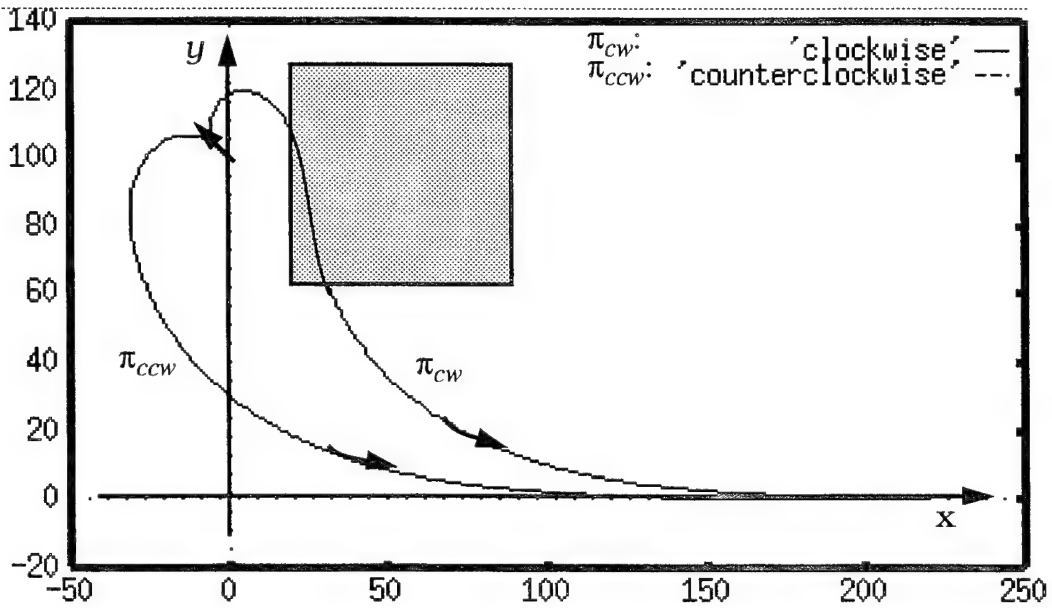


Figure A.4: Clockwise and Counterclockwise Tracking Trajectory

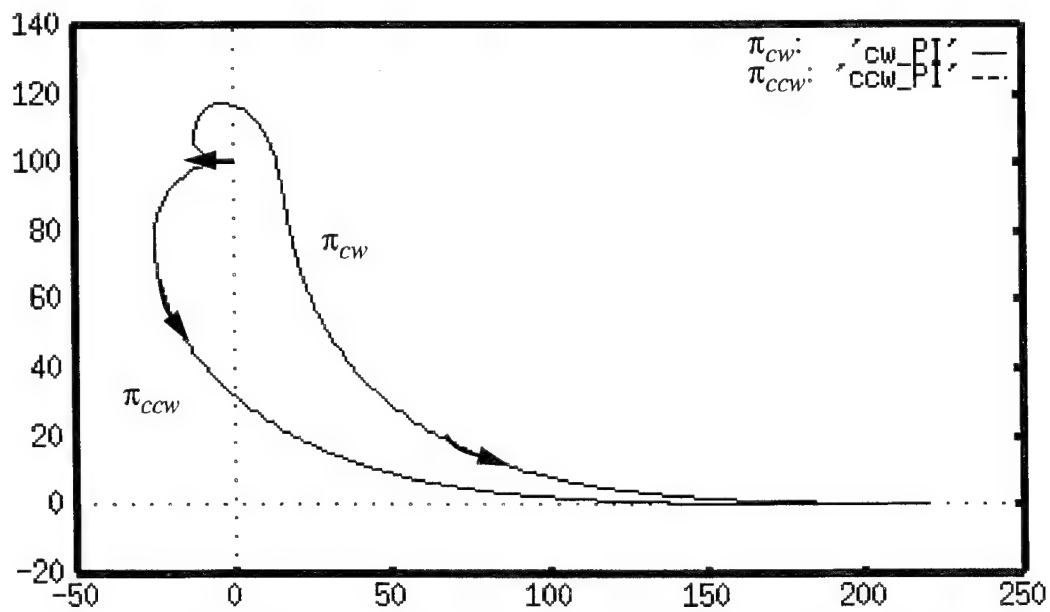


Figure A.5: Clockwise and Counterclockwise Tracking Trajectory with Initial $\Delta\theta = \pi$

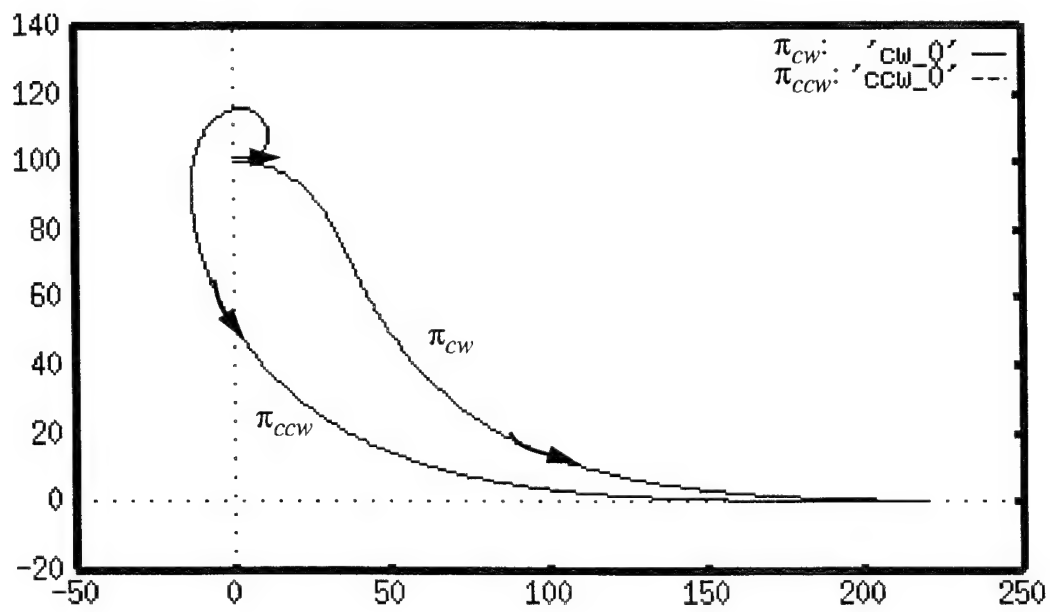


Figure A.6: Clockwise and Counterclockwise Tracking Trajectory with Initial $\Delta\theta = 0$

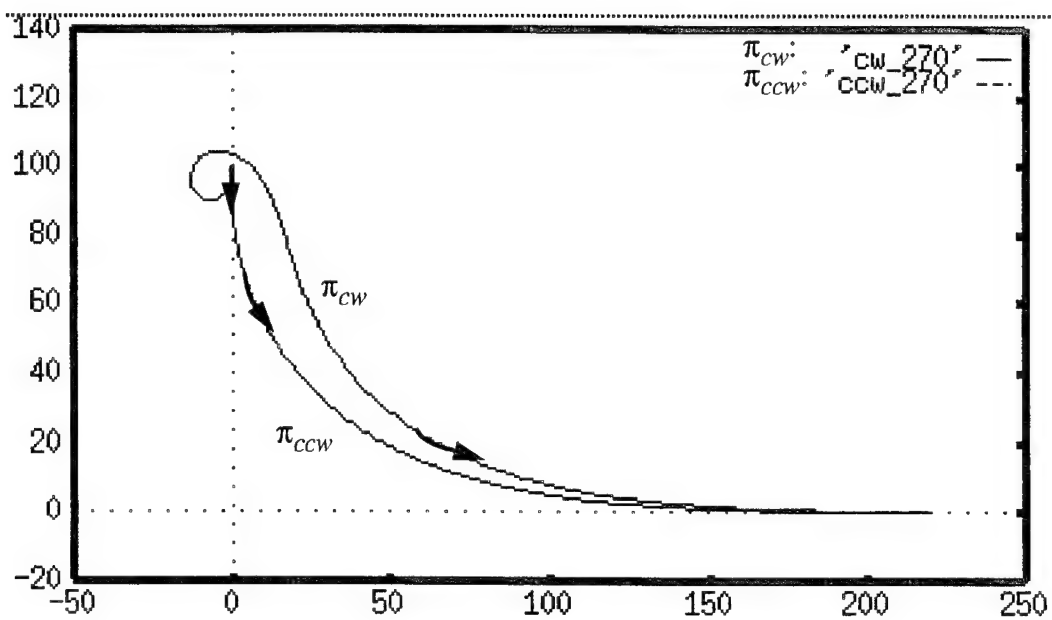


Figure A.7: Clockwise and Counterclockwise Tracking Trajectory with Initial $\Delta\theta = -\pi/2$

APPENDIX B. DYNAMIC MEMORY MANAGEMENT STRATEGY

A. INTRODUCTION

The dynamic memory management involves allocation and deallocation of memory blocks with requested size. If they are not handled properly, the memory fragmentation will eventually cause inefficient memory use. Without any management strategy, we may allocate memory blocks to the user sequentially as request and deallocate them in the order of deallocation request. For example, if we have 100 k bytes memory available for dynamic memory use, and the request for the memory are:

- a. Request a block of size 20 k.
- b. Request a block of size 30 k.
- c. Request a block of size 15 k.
- d. Request a block of size 25 k.
- e. Release the block in request b.
- f. Request a block of size 25 k.
- g. Request a block of size 10 k.
- h. Release the block in request a.
- i. Request a block of size 10k.
- j. Release the block in request c.
- k. Release the block in request d.
- l. Request a block of size 20 k.
- m. Request a block of size 20 k.

The request **a- d** will result the memory partitioned as Figure B.1. There is only one memory block free with size 10 k. The request **e-m** will result the memory partitioned as Figure B.2. There are several free blocks with total size 35 k after request l. However, when request **m** comes, system will not be able to allocate any memory block to this request, even though the size of total available free memory is large than the size of current request.

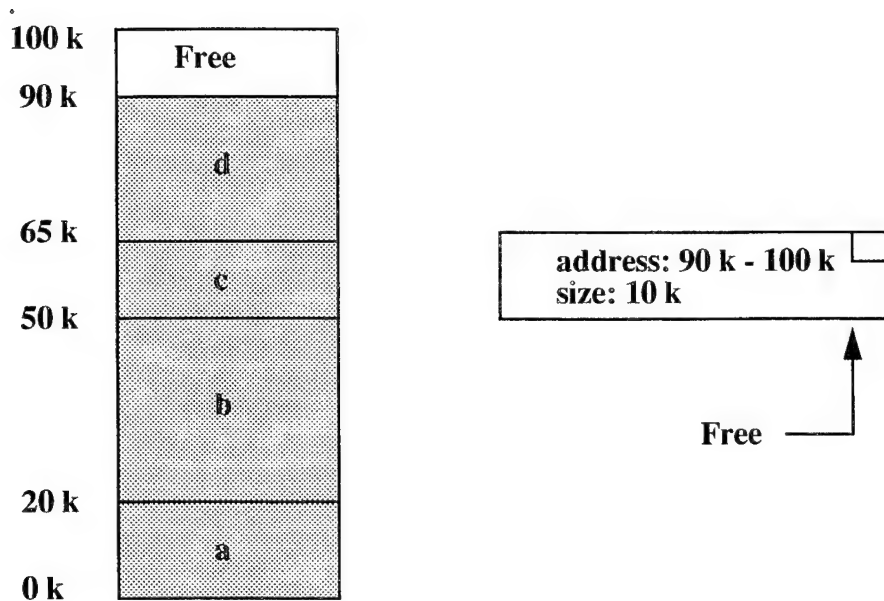


Figure B.1: Dynamic Memory Allocation Partition after Request a-d

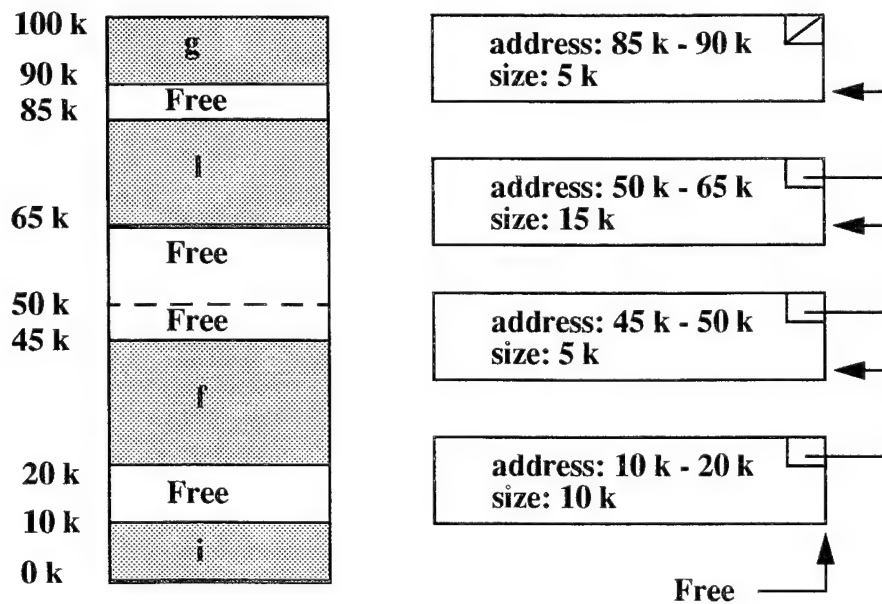


Figure B.2: Dynamic Memory Allocation Partition after Request e-m

After more allocation and deallocation operations, the dynamic memory space may be partitioned into many small blocks either consecutive or separate. This is not desirable because it will eventually make it impossible to fill the legitimate request of a user needing one larger memory block. The request **m** is an example. To remedy this problem, we must have a method that allows a block being returned to the available list to be coalesced with any other block(s) in the list that are the physical neighbor(s) of the returning block. Therefore a buddy system, which can coalesce neighboring free blocks to form a larger memory block available, is needed for dynamic memory allocation management.

B. BUDDY SYSTEMS

The general purpose of a buddy system is to provide a method that designates one or two buddy blocks for each block when partitioning the memory. The buddy blocks have to reside right next to its corresponding block. When a block is to be released, its buddy block will be checked to see if it's available for coalescence. If the coalescence is possible, the two blocks will be returned as one larger block to available memory. The coalescence will be performed recursively whenever buddy block is available. To determine the buddy of a given block, it is necessary to impose certain restrictions on block sizes and/or to store a fair amount of bookkeeping information in each block.

The most common buddy systems are [28]:

- Binary buddy system.
- Fibonacci buddy system.
- Boundary tag buddy system.

Among them, the Fibonacci buddy system is the most popular one for the following reason. Firstly it allows for a greater variety of possible block sizes in a given amount of memory than Binary buddy system. Second, the Fibonacci buddy system provides an efficient method to allocate and deallocate memory blocks.

The basic requirement of buddy system is that any block size (except the smallest one) must result from coalescing two smaller blocks, so that the addresses of the buddies

can be easily computed in allocation and deallocation operation. In Fibonacci sequence, the i th member of the sequence is recursively defined as

$$F_i = F_{(i-1)} + F_{(i-2)} \quad \text{for } i > 2$$

Therefore using Fibonacci sequence as the memory splitting strategy meet the requirement of buddy system. The Fibonacci sequence can be stored in an array of desired data structure which is used to indicate the free memory blocks with size corresponding to the Fibonacci number. In the array, each element may contain a pointer that points to the available block of corresponding size.

Let's take an example to demonstrate how the Fibonacci buddy system works. Suppose we have $F_1 = 1$, $F_2 = 2$, then the members of the Fibonacci sequence are:

1, 2, 3, 5, 8, 13, 21, 34, ...

And suppose that we are managing 21 k memory and are faced with the following requests for storage:

- a. Request for 7k.
- b. Request for 7k.
- c. Request for 2k.
- d. Release 7k of request b.
- e. Release 2k of request c.
- f. Release 7k of request a.

Figure B.3 to B.9 illustrate the allocation, deallocation and resulting coalescing that would occur as these requests were processed. In each figure, the rectangle in the center illustrates the memory block(s) status. The blank block means it is a free memory block while the shaded block indicates an occupied space. On the left, a tree with a number in each circle illustrates the splitting of Fibonacci sequence after an allocation or a deallocation request is processed starting with the size of entire available memory space. Each node of leaves of the tree corresponds to a memory block. On the right, the array uses pointers that point to free memory blocks of corresponding size.

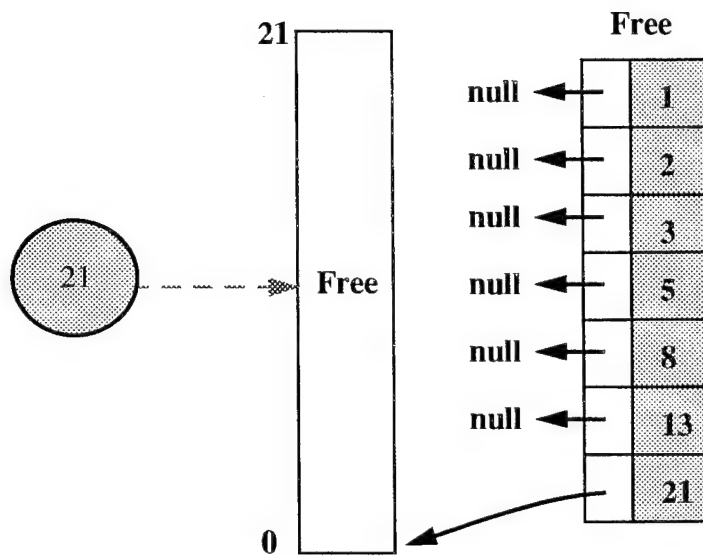


Figure B.3: Dynamic Memory Allocation Partition, Initial State of the Memory

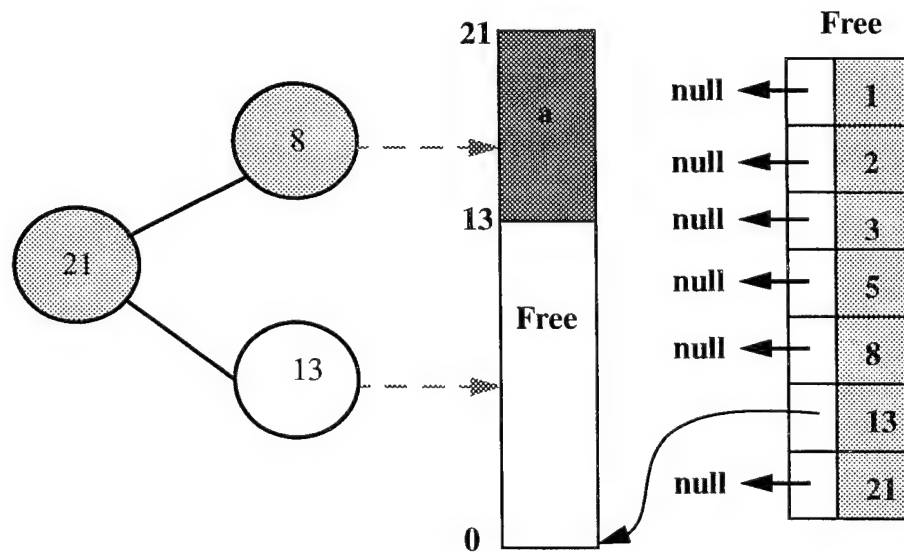


Figure B.4: Dynamic Memory Allocation Partition after Request a Processed - 8k Allocated

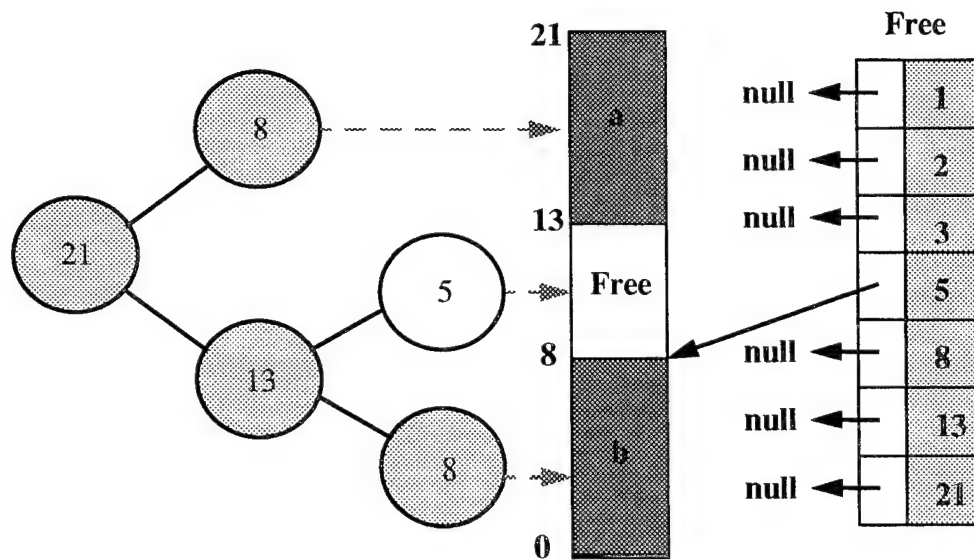


Figure B.5: Dynamic Memory Allocation Partition after Request (b) Processed - 8k Allocated

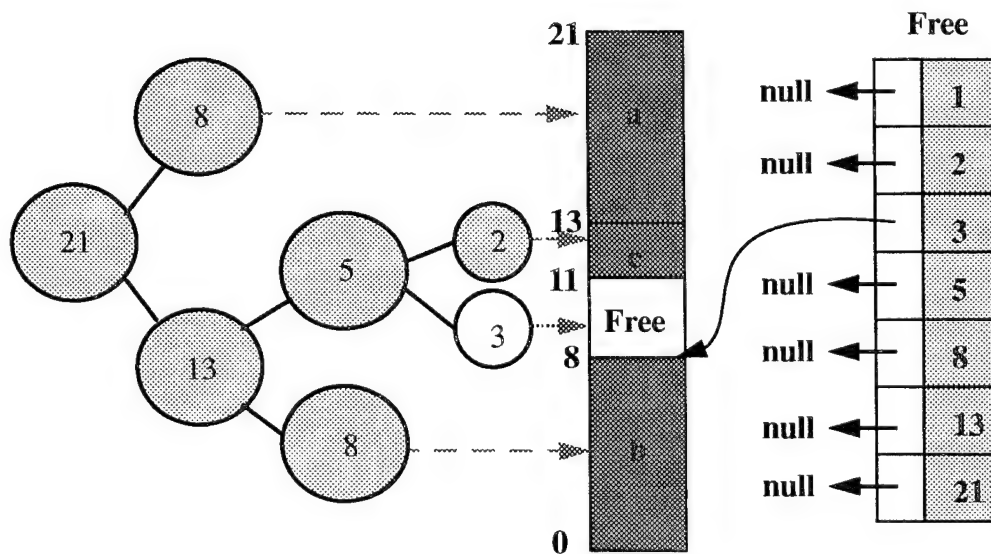


Figure B.6: Dynamic memory allocation Partition after Request (c) Processed - 2k Allocated

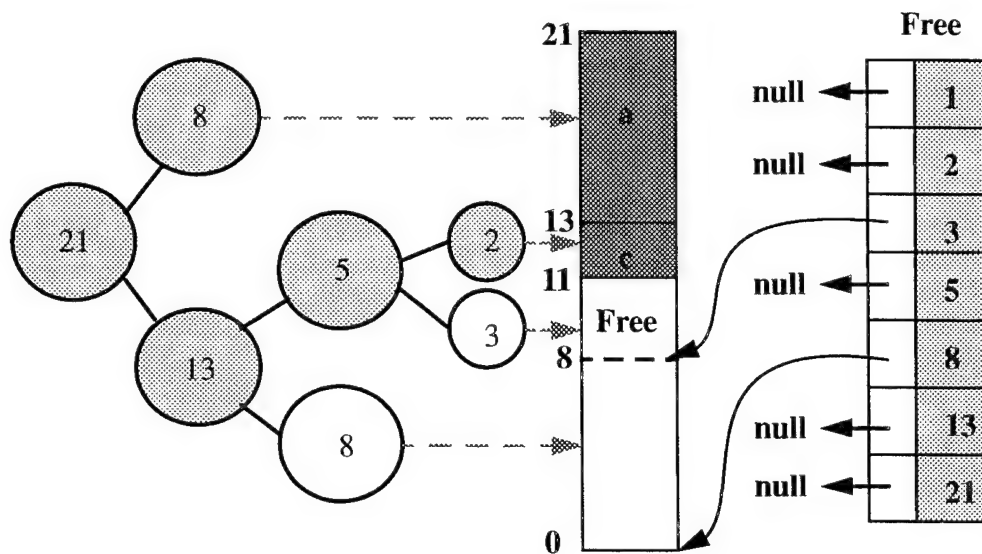


Figure B.7: Dynamic Memory Allocation Partition after Request (d) Processed - 8k Deallocated But No Coalescing

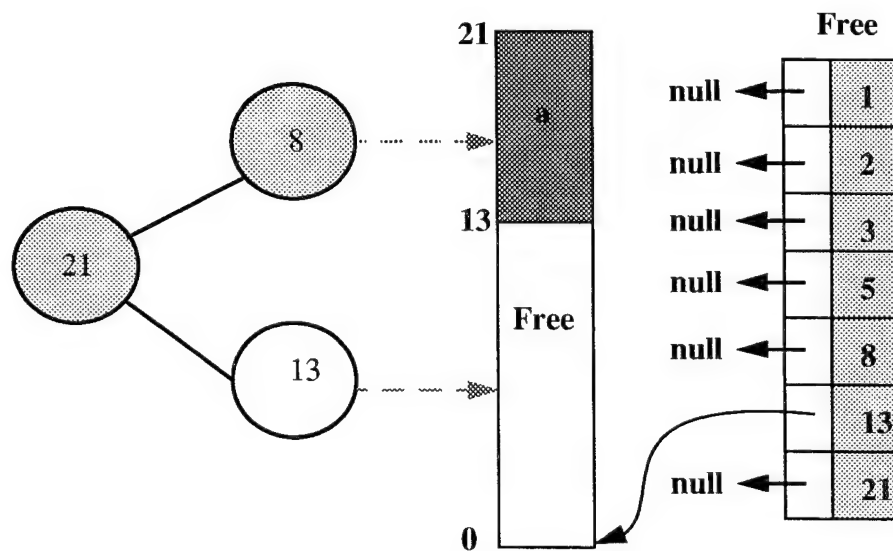


Figure B.8: Dynamic Memory Allocation Partition after Request (e) Processed - 2 k Deallocated and Coalescing Occurs Up to 13 k Block

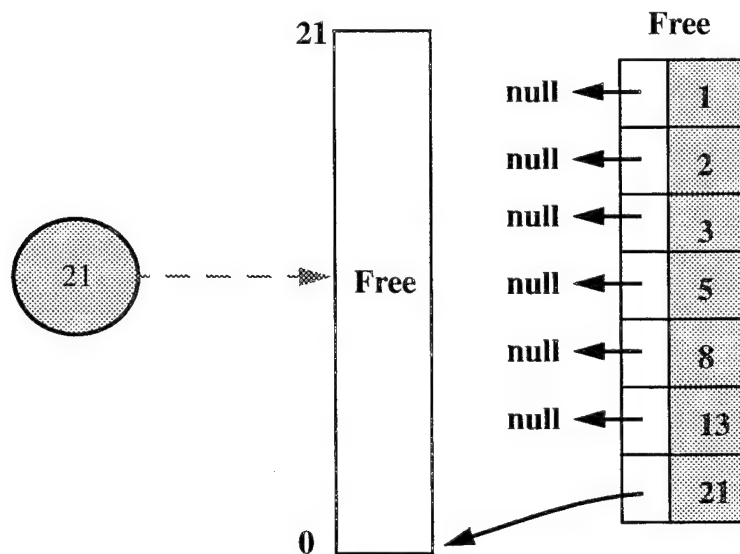


Figure B.9: Dynamic Memory Allocation Partition after Request (f) Processed - 8 k Deallocated and Coalescing Occurs - One Free 21 k Block Results

Naturally, some additional overhead is required when the Fibonacci buddy system is used. First, depending on the implementation scheme, it may be necessary to store the Fibonacci numbers themselves in an array to allow quick access to data necessary to allocate, split, and coalesce blocks. Second, it is necessary to store some bookkeeping data within each block so that it tells the block's status (free or not), size, links to other blocks, and depth of being left buddy to other blocks as Figure B.10.

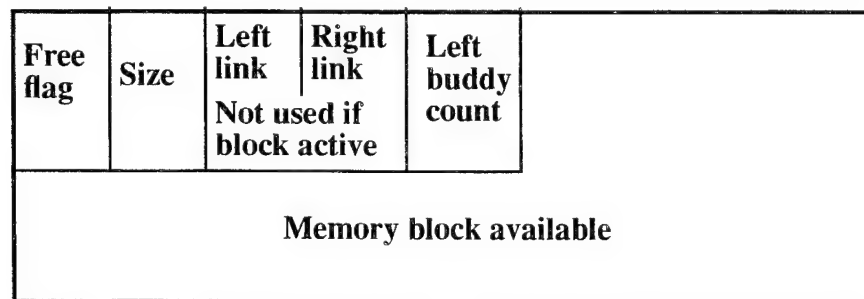


Figure B.10: Bookkeeping Information in Block for Fibonacci Buddy System

APPENDIX C. USER PROGRAM EXAMPLES

```
*****
Program Name : user.c-#1
Purpose      : For Model Based Motion Planning demo.
Parameters   : void
Returns      : void
Comments     : 4/1/95 Chien-Liang Chuang
*****
```

```
#include "user.h"
```

```
void          setconfig();
CONFIGURATION chooseConfig();
CONFIGURATION q, q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10,
              q11, q12, q13, q14, q15, q16, q17, q18, goal;
```

```
void
user()
{
```

```
    int      go;
    double    theta;
    PATH_ELEMENT path;
```

```
    createModel();
```

```
    go = 1;
    setconfig();
```

```
    printf("\nSelect 0-18 for default START,99 for input: ");
```

```
    q = chooseConfig();
```

```
    setRobotConfigImm(q);
```

```
    printf("\nSelect 0-18 for default GOAL, 99 for input : ");
```

```
    goal = chooseConfig();
```

```
    setLinVelImm(30.0);
```

```
    MotionLog(NULL, 20, 0);
```

```
    while (go) {
```

```
        gotoImm(goal);
```

```
        path = getPathElement();
```

```

        while (path.pathType.mode == STOPMODE)
            path = getPathElement();

        while (path.pathType.mode != STOPMODE)
            path = getPathElement();

    q = getRobotConfig();

    printf("\n\nGo to somewhere else? 1/0");

    go = GetInt();

    if (go != 0)
    {
        printf("\nSelect 0-18 for GOAL,99 for input : ");
        goal = chooseConfig();
    }
}

```

```

/*****
Function : setconfig() to set default configuration
*****/
void
setconfig()
{
    q0 = defineConfig(200.0, 214.0, 0.0, 0.0);
    q1 = defineConfig(125.0, 345.0, HPI, 0.0);
    q2 = defineConfig(50.0, 657.0, PI, 0.0);
    q3 = defineConfig(125.0, 872.0, HPI, 0.0);
    q4 = defineConfig(125.0, 2000.0, -HPI, 0.0);
    q5 = defineConfig(50.0, 1862.0, PI, 0.0);
    q6 = defineConfig(50.0, 2148.0, PI, 0.0);
    q7 = defineConfig(50.0, 2433.0, PI, 0.0);
    q8 = defineConfig(190.0, 1980.0, 0.0, 0.0);
    q9 = defineConfig(190.0, 1390.0, 0.0, 0.0);
    q10 = defineConfig(555.0, 867.0, 0.0, 0.0);
    q11 = defineConfig(800.0, 743.0, 0.0, 0.0);
    q12 = defineConfig(707.0, 720.0, -HPI, 0.0);
    q13 = defineConfig(417.0, 720.0, -HPI, 0.0);
    q14 = defineConfig(417.0, 546.0, -HPI, 0.0);
    q15 = defineConfig(706.0, 546.0, -HPI, 0.0);
    q16 = defineConfig(-93.0, 2138.0, PI, 0.0);
    q17 = defineConfig(-278.0, 2121.0, PI, 0.0);
    q18 = defineConfig(-505.0, 2150.0, PI, 0.0);
}

```

```

/*****
Function : chooseConfig() for selecting a configuration
*****/
CONFIGURATION
chooseConfig()
{
    int      set;
    double    x, y, t;
    CONFIGURATION q;

    set = GetInt();

    switch (set){

        case 0: q = q0; break;
        case 1: q = q1; break;
        case 2: q = q2; break;
        case 3: q = q3; break;
        case 4: q = q4; break;
        case 5: q = q5; break;
        case 6: q = q6; break;
        case 7: q = q7; break;
        case 8: q = q8; break;
        case 9: q = q9; break;
        case 10: q = q10; break;
        case 11: q = q11; break;
        case 12: q = q12; break;
        case 13: q = q13; break;
        case 14: q = q14; break;
        case 15: q = q15; break;
        case 16: q = q16; break;
        case 17: q = q17; break;
        case 18: q = q18; break;

        default :
            printf("\nInput configuration : ");
            printf(" x =");
            x = GetReal();
            printf(" y =");
            y = GetReal();
            printf(" theta =");
            t = GetReal();
            t = t * DAR;
            q = defineConfig(x, y, t, 0.0);
            break;
    }
    return(q);
}

```

```

*****
Program Name : user.c-#2
Purpose      : For Sonar Testing
Parameters   : void
Returns      : void
Comments     : 4/1/95 Chien-Liang Chuang
*****

```

```

#include "user.h"

#define FREQ 1

int SONARNUM, DATATYPE;
int type;

void user1();
void user2();
void user3();

void user()
{
    int selection;

    printf("\n Enter 1 for Stationary testing ");
    printf("\n Enter 2 for Moving (line) testing ");
    printf("\n Enter 3 for Moving (rotate) testing ");
    selection=GetInt();

    printf("\n Input sonar # : ");
    SONARNUM=GetInt();

    printf("\n Input 1 for logging RAW-data, 2 for logging GLOBAL-data : ");
    type=GetInt();
    if (type==1)
        DATATYPE = SONAR_RAW;
    else
        DATATYPE = SONAR_GLOBAL;

    switch (selection)
    {
        case 1:
            user1();
            break;
        case 2:
            user2();
            break;
        case 3:
            user3();
            break;
        default:

```



```

        break;
    }
}

```

```

/*****
Function : user1() for stationary testing
*****/
void user1()
{
    double distance;
    int cnt = 0;

    EnableSonar(SONARNUM);

    SonarLog(FREQ,0,SONARNUM,DATATYPE);

    waitMS(30);

    while (++cnt <= 50)
    {
        distance = Sonar(SONARNUM);

        printf("\nSonar Range is %f",distance);

        waitMS(50);
    }
}

```

```

/*****
Function : user2() for moving (line) testing
*****/
void user2()
{
    double dist, speed;
    CONFIGURATION q, p;

    printf("\n Input desired speed : ");
    speed = GetReal();
    setLinVelImm(speed);

    printf("\n Input traveling distance : ");
    dist = GetReal();

    p = defineConfig(0.0, 0.0, 0.0, 0.0);

```

```

q = defineConfig(dist, 0.0, 0.0, 0.0);
setRobotConfig(p);
EnableSonar(SONARNUM);
SonarLog(5,0,SONARNUM,DATATYPE);
bline(q);
}

```

```

/*****
Function : user3() for moving (rotate) testing
*****/
void user3()
{
    CONFIGURATION q;

    q = defineConfig(0.0, 0.0, 0.0, 0.0);

    setRobotConfig(q);

    EnableSonar(SONARNUM);

    SonarLog(FREQ,0,SONARNUM,DATATYPE);

    waitMS(30);

    Rotate(PI);
}

```

LIST OF REFERENCES

1. Latombe, J. C., Robot Motion Planning, Kluwer Academic Publishers, Norwell, MA, 1991.
2. Doyle, A. B., and Tones, D. I., "A Tangent Based Method for Robot Path Planning", IEEE int. Conf. on Robotics and Automation, pp. 1561-1566, 1994.
3. Hwang, Y. K. and Ahuja, Narendra, "Gross Motion Planning -- A Survey", ACM Computing Surveys, Vol. 24, No. 3, September 1992.
4. Aurenhammer, F., "Voronoi Diagrams--A Survey of Fundamental Geometric Data Structure". ACM Comput. Surv. 23, 3, pp. 345-405, 1991.
5. Tan, Liek Foo, "Motion Planning for Rigid Body Robots", Master's Thesis, Naval Postgraduate School, Monterey, California, June 1992.
6. Canny, J. F., and Lin, M. C., "An Opportunistic Global Path Planner", IEEE int. Conf. on Robotics and Automation, pp. 1554-1561, 1990.
7. Kovalchik, J., "Layered Motion Planning for Autonomous Mobile Robots Using a Steering Function", PhD dissertation, Naval Postgraduate School, Monterey, California, 1995.
8. Warren, C. W., "Fast Path Planning Using Modified A* Method", IEEE int. Conf. on Robotics and Automation, pp. 662-667, 1993.
9. Vacherand F., "Fast Local Path Planner in Certainty Grid", IEEE int. Conf. on Robotics and Automation, pp. 2132-2137, 1994.
10. Koditschek, D. E., "Robot Planning and Control via Potential Functions. In Robotics Review", vol. 1, O. Khatib, J. Graig, and T. Lozano-Perez, Eds. MIT Press, Cambridge, Mass, 1989.
11. Hwang, Y. K., Chen, P. C., Maciejewski, A. A., and Neidigk, D.D., "A Global Motion Planner for Curve-Tracing Robots", IEEE int. Conf. on Robotics and Automation, pp. 2-7, 1994.
12. Jacobs, P. and Canny, J. "Planning Smooth Paths for Mobile Robots", IEEE int. Conf. on Robotics and Automation, pp. 2-7, 1994.
13. Laumond, L., "Feasible Trajectory for Mobile Robots with Kinematic and Environment Constraints", *Intelligent Autonomous Systems*, pp. 346 - 354, 1986.
14. Vasseur, H. A., Pin, F. G., and Taylor, J. R., "Navigation of a Car-like Mobile Robot Using a Decomposition of the Environment in Convex Cells", IEEE int. Conf. on Robotics and Automation, pp. 1496-1502, 1991.

15. Fraichard, T. and Laugier, C., "Path-Velocity Decomposition Revisited and Applied to Dynamic Trajectory Planning", IEEE int. Conf. on Robotics and Automation, pp. 40-45, 1993.
16. Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach", *IEEE Transaction on Computer*, vol. C-32, no. 2, pp. 108-119, Feb. 1983.
17. Adams, J. F., "Algebraic Topology: A Student's Guide" Cambridge Univ. Press, London and New York, 1972.
18. Kanayama, Y. , "Two Dimensional Wheeled Vehicle Kinematics", IEEE int. Conf. on Robotics and Automation, pp. 3079-3084, 1994.
19. Kanayama, Y. , "Introduction to Motion Planning", Lecture note, Naval Postgraduate School, 1995.
20. Kanayama, Y. and Krahn, G. W., "Theory of Two-Dimensional Transformations", Lecture note, Naval Postgraduate School, 1994.
21. Macpherson, D. L., "Automated Cartography by An Autonomous Mobile Robot Using Ultrasonic Range Finders", Ph. D. dissertation, Naval Postgraduate School, Monterey, California, Sept. 1993.
22. Kanayama, Y., Kovalchik, J., Chuang C. and Kelbe, F., "Motion Planning: for Autonomous Mobile Robots", Autonomous Vehicle Minecountermeasure Symposium Proceedings, April 1995.
23. Manber, Udi., *Introduction to Algorithms*, Addison-Wesley Publishing Company, Reading, MA , pp. 273-277, 1989.
24. Kanayama, Y., Kimura, K., Miyazaki, F. and Noguchi, T., "A Stable Tracking Control Method for an Autonomous Mobile Robot", IEEE int. Conf. on Robotics and Automation, pp. 1315-1317, 1988.
25. Kanayama, Y. and Onishi, M., "Locomotion Functions for a Mobile Robot Language, MML", IEEE int. Conf. on Robotics and Automation, pp. 1110-1115, 1991.
26. Ironics, Inc., IV-SPARC-25A/33A VMEbus Single Board Super Computer and MultiProcessing Engine -- User's Manual, Ironics, Inc., New York, 1992.
27. Weaver, D., and Germond, T., *The SPARC Architecture Manual*, PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.
28. Naps, T. L., Nace, D. W. and Singh, B., *Introduction to Computer Science: Programming, Problem Solving and Data Structure*, West Publishing Company, St. Paul, MN, pp. 914-930, 1992.

BIBLIOGRAPHY

Arnold, V.I., *Mathematical Methods of Classical Mechanics*, Springer-Verlag, New York, 1978.

Barbehenn, M., and Hutchinson, S., "Efficient Search and Hierarchical Motion Planning Using Dynamic Single-Source Shortest Paths Trees", *IEEE int. Conf. on Robotics and Automation*, pp. 566-571, 1993.

Bessonnet, G. and Lallemand, J.P., "Planning of Optimal Free Paths of Robotic Manipulators with Bounds on Dynamic Forces", *IEEE int. Conf. on Robotics and Automation*, pp. 270-275, 1993.

Bui, X., Boissonnat, J. D., Soueres, P., and Laumond, J. P., "Shortest Path Synthesis for Dubins Non-holonomic Robot", *IEEE int. Conf. on Robotics and Automation*, pp. 2-7, 1994.

Cameron, J. M., and Book, W. J., "Optimal Path Planning for the Motion of a Wheel", *IEEE int. Conf. on Robotics and Automation*, pp. 1574-1579, 1994.

Campion, G., Bastin, G. and D'Andrea-Novet, B., "Structural Properties and Classification of Kinematic and Dynamic Models of Wheeled Mobile Robots", *IEEE int. Conf. on Robotics and Automation*, pp. 462-469, 1993.

Challou, D.J., Gini, M. and Kumar, V., "Parallel Search Algorithms for Robot Motion Planning", *IEEE int. Conf. on Robotics and Automation*, pp. 46-51, 1993.

Chazelle, B., "Approximation and Decomposition of Shapes", *Algorithmic and Geometric Aspects of Robotics*, Lawrence Erlbaum Associates, Hillsdale, NJ. 1987.

Fluery, S., Soueres, P., and Laumond, J., "Primitives for Smoothing Mobile Robot Trajectory", *IEEE int. Conf. on Robotics and Automation*, pp. 832-839, 1993.

Fraichard, Th. and Scheuer, A., "Car-Like Robots and Moving Obstacles", *IEEE int. Conf. on Robotics and Automation*, pp. 64-69, 1994.

Fujimura, K. and Samet, H., "Time-Minimal Paths among Moving Obstacles", *IEEE int. Conf. on Robotics and Automation*, pp. 1110-1115, 1994.

Gray, B., "Homotopy Theory: An Introduction to Algebraic Topology", The Academic Press, 1975.

Kanayama, Y., "Least Cost Paths with Algebraic Cost Functions", *IEEE int. Conf. on Robotics and Automation*, pp. 75-80, 1988.

Kanayama, Y. and Hartman, B., "Smooth Local Path Planning for Autonomous Vehicles", IEEE int. Conf. on Robotics and Automation, pp. 1265-1270, 1989.

Kanayama, Y. and Noguchi, T., "Locomotion Functions in the Mobile Robot Language," Proc. IEEE/RSJ International Workshop on Intelligent Robots and Systems, pp. 542-549, 1989.

Kanayama, Y. , "Vehicle Motion Planning under Geometric Uncertainty", Proposal of research to NSF, 1994.

Keil, J. M. and Sack, J. R., "Minimum Decomposition of Polygonal Objects", *Computer Geometry*, Elsevier Science Publishers, North Holland, Amsterdam, pp. 197-216 1985.

Samuel, S., and Keerthi, S. S., "Numerical Determination of Optimal Non-holonomic Paths in the Presence of Obstacles", IEEE int. Conf. on Robotics and Automation, pp. 826-831, 1993.

Shiller, Z., Serate, W. and Hua, M., "Trajectory Planning of Tracked Vehicles", IEEE int. Conf. on Robotics and Automation, pp. 796-801, 1993.

Stappen, A. F., Halperin, D., and Overmars, M. H., " Efficient Algorithms for Exact Motion Planning amidst Fat Obstacles", IEEE int. Conf. on Robotics and Automation, pp. 297-304, 1993.

Wilfong, G. "Motion Planning for an Autonomous Vehicle", IEEE int. Conf. on Robotics and Automation, pp. 529-533, 1988.

Wilfong, G. "Shortest Path for Autonomous Vehicles", IEEE int. Conf. on Robotics and Automation, pp. 15-20, 1989.

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 013
Naval Postgraduate School
Monterey, California 93943-5101 | 2 |
| 3. | Chairman, Department of Computer Science
Code CS
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 4. | Professor Yutaka Kanayama, Code CS/Ka
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943 | 2 |
| 5. | Professor Man-Tak Shing, Code CS/Sh
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 6. | Professor Thomas Wu Code CS/Wq
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 7. | Professor Craig Rasmussen, Code MA/Ra
Department of Mathematics
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 8. | Professor Ranjan Mukherjee, Code ME/Mk
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 9. | Mrs. Chien-Liang Chuang
7-1, Lane 285, Ching-Shing Street
Chung-Ho, Taipei , Taiwan, R.O.C. | 3 |

10. Mr. Joseph G. Kovalchik
113 Roosevelt Street
Edwardsville, Pennsylvania 18704

1